An Experimental Evaluation of Auto-exploratory, Average-reward Reinforcement Learning

by
Kimberly Mach

A PROJECT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented April 6, 2004
Commencement June 2004

©Copyright by Kimberly Mach

April 6, 2004

All rights reserved

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

Reinforcement Learning (RL) is the type of artificial intelligence where an agent learns through numerical rewards as a consequence of its actions. A reinforcement learner aims to maximize the overall reward that it receives. Since most environments of interest involve some level of stochasticity, accomplishing this task is not trivial. The issue of what optimization criteria will be used is an important one. Discounted total reward and average reward are two main choices. In this project, we will use the average reward optimization, which amounts to maximizing the average reward received per time step.

Dynamic Programming (DP) can be used to learn an optimal solution to a given problem by propagating reward information backwards to other states [4], and is is central to the theory behind many reinforcement learning algorithms. Dynamic programming is typically used in assigning values, or a measure of desirability, to every state or state-action pair, and is essential for detection of the true value function.

RL methods based on dynamic programming can learn through the use of discounted or average reward. Policies, a mapping of states to actions, are developed and evolved through the use of either value or policy iteration and the examination of the values these methods produce. Value iteration can be summarized as a method of continuously updating state values in a matrix until these numbers have converged (or, at least, until they start changing at a slower rate). There are several styles of value iteration, including the asynchronous, online varieties. One difference between reinforcement learning and dynamic programming is that DP is given the action models and RL is not.

H-Learning [21] is a model-based average reward reinforcement learning. Models of the transition probability and of the rewards received are maintained along with the state value table. R-Learning [17] is a model-free, average-reward, reinforcement-learning algorithm. This means that models of the problem are not directly learned, rather a table of state-action values is learned instead to directly approximate the policy.

An issue that is critical to all reinforcement learning algorithms is that of exploration. It is of utmost importance for the agent to fully explore its environment so that it may make informed decisions on what states it would prefer to be in. However, the agent must also use what values it does know and, through the use of its learning algorithm, make logical decisions on what action to take next–otherwise, the agent would be as informed as a random walker. In auto-exploratory reinforcement learning the agent automatically searches the state space, while always taking greedy actions. This is done through the strategy of optimism under uncertainty [18]; the value of unexplored regions are kept higher than previously examined regions, thus forcing exploration.

Of particular interest in this project are the auto-exploratory versions of R- and H-Learning (AR- and AH-Learning). These average-reward methods force exploration by making use of a parameter ρ, which represents the aspired average reward per step and is maintained at a value higher than the actual average reward per step. This causes a reduction in the values of explored state-action pairs, and encourages the agent to explore other states.

The size of the state space involved in an artificial intelligence problem can grow to unacceptable proportions, and this gives voice to the need to shrink the size of the value function and the policy. Function approximation is the process of approximating the value function of an environment in such a way as will

shrink its size. The method that is used in this paper is piecewise linear function approximation using gradient descent.

We compare table-based and function-approximated versions of the auto-exploratory methods AH- and AR-Learning. This is in response to the findings in [22] where AR-Learning was unable to learn a policy as high as the one learned by AH-Learning. We find AR-Learning to be comparable to AH-Learning in the table-based domain. We also find AR- and AH-Learning to be comparable in most function-approximated domains; however, in a few smaller environments they achieve noticeably smaller results than their table-based counterparts. We hypothesize that this discrepancy is due to a combination of the representation of linear function approximation that was used and the coarse depiction of state features.

The rest of the paper is organized as follows: We explore first an extensive background of reinforcement learning in section 2. In section 3 we move to auto-exploratory theory and methods. Section 4 includes a general description of gradient descent function approximation, featuring the methods of linear function approximation (used in this paper) and neural networks. The environment used in this project is described in section 5, and the particular application of function approximation that was used is outlined in section 6. Results are presented in section 7 and this paper is summarized in section 8.

2. BACKGROUND

Many AI problems can be described in the framework of Markov Decision Problems which are characterized by a set of states that the agent finds itself in and a set of actions the agent can take. The program that is being used to implement the specific artificial intelligence algorithm and to solve the problem at hand can be referred to as an *agent*. The *state*, s, of the environment that the agent is currently in refers to everything in the environment which might be of importance to the agent within the program (for instance, if playing a chess game, the state might refer to positions of the pieces on the board). The set of all possible states in the environment which the agent is working in is S. At any given time, the agent can take an *action*, a (which is in the set of all legal actions from the current state, $A(s)$) that will alter the environment by taking the agent to a new state, s'. Every time that an agent takes an action, it receives an immediate reward, r. When necessary to prevent confusion, s_t, a_t, and r_t will be used to describe the state the agent is in, the action the agent is taking, or the reward the agent received, at time t. A mapping of states to actions, or, rather, the decisions of what actions to take in each state, is called a policy, π. The overall goal of an artificial intelligence program is for the agent to act in such a manner as to optimize some criterion (based on the rewards, and to be discussed later in more detail) and as a result learn an optimal policy, π^*. [1]

The rest of this section will provide theoretical background to properly introduce the problem of reinforcement learning.

---

[1]The notation used here will be used throughout the remainder of this paper.

2.1. Markov Decision Processes

In this paper, we will restrict ourselves to dealing only with Markov Decision Problems (MDP). An MDP consists of a set of states S and a set of actions A from which the agent can choose. In any time step t the agent observes the state s, takes action a, receives reward r, and ends in a new state s'. It is important to mention: even though they are represented as a singular event, a, actions may involve altering the state of the environment in several different ways at once [2]. For example, in a game of checkers, one move (or action a) could involve jumping over the opponent's piece. The resulting action would involve changing the location of the agent's piece and removing the opponent's piece from the board.

An important property of an MDP is that the future evolution of the world is completely determined by the current state (no explicit description of the past history needs to be kept to determine the best course of action). This property is known as the Markov property. Mathematically, one can state this as:

$$\begin{aligned} P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0, r_0) \\ = P(s_{t+1} = s', r_{t+1} = r | s_t, a_t), \end{aligned}$$

(where time is quantized, and P is the probability [20]). In this paper, we also assume that MDP's have a finite set of states and actions.

2.2. Policies and Optimization Criteria

What exactly does an agent classify as a good action? It is mentioned above that for every action taken there is a reward given to the agent. Most environments of interest involve an element of stochasticity (randomness), such that the immediate reward for any one action will not remain constant throughout the

entire lifetime of an agent. How do we figure out which state is the best to be in, if this stochasticity changes the rewards received in any given state? Stochasticity and delayed feedback are two main considerations. To briefly expound on the idea of delayed feedback, consider the game of chess. In this environment, feedback is received only at the end of a game. The problem here is how the agent will know what move to start with, when the only reward in this environment is based on whether or not the agent won. This is the delayed feedback problem.

A policy is a mapping from states to actions. The goal of a policy is to optimize not just short-term, but the long-term rewards. If this goal is fulfilled, the agent will not simply take actions that promise immediate rewards, but will take instead those actions that return the highest total reward in the long run. The expected total return that an agent may anticipate when following a policy π is the value of that policy. The return of π is defined by one of the following optimization criteria.

2.2.1. Total versus Discounted Reward

In domains that have a natural final state that will eventually be reached, one would like to maximize the sum of all rewards received up to the final (or absorbing) state, that is:

$$R_T = r_0 + r_1 + r_2 + \cdots + r_\tau,$$

where τ is the time to reach the absorbing state, 0 is the time of the start of the program, and R_T is the total reward received until the absorbing state is reached (the quantity that we are maximizing). Since the rewards received are random variables, one might optimize the *expectation* of the above sum, or:

$$R_T = E[\sum_{t=0}^{T} r_t],$$

where E is the expectation of the amount enclosed in the brackets, and T is the total time the program is run.

If the agent lives in a world with unbounded time, we run into another problem–the total reward, R_T, becomes infinite. One alternative we have is to use *discounting*, which will take a small number, γ, called a discount factor, where $0 < \gamma \leq 1$ and multiply each successive reward received by a corresponding power of γ, such as:

$$R_T = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots + \gamma^t r_t + \cdots = E[\sum_{t=0}^{\tau} \gamma^t r_t]. \tag{2.1}$$

This keeps the total reward's sum from blowing up to infinity. Discounted reinforcement learning, though convenient, has the disadvantage that it tends to prioritize immediate rewards and ignore/subdue the value of rewards in the distant future [20, 21, 11].

2.2.2. Optimization by Average Reward

Average-reward reinforcement learning is an undiscounted method that works by concentrating not on the immediate rewards offered (as discounting does), but rather on the long-term average of the total reward received. In this way, average-reward reinforcement learning takes on the 'big picture' instead of giving undue importance to the immediate returns. Mathematically, this is done by maximizing:

$$\rho = \lim_{n \to \infty} \frac{1}{n} \sum_{t=0}^{n} E[r_t],$$

where ρ is the gain, or the average reward.

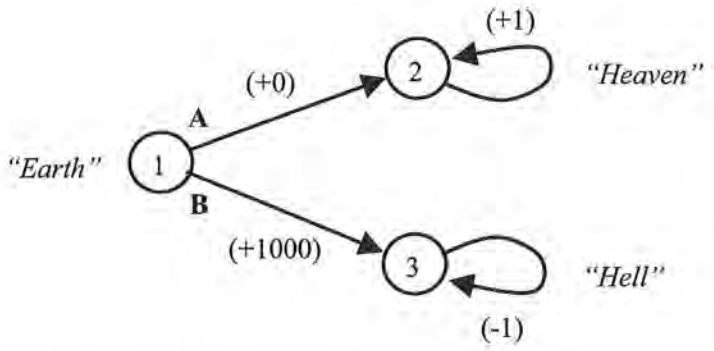


FIGURE 2.1. An example where taking the average reward will yield the proper answer, but discounting will not (from A. Schwartz).

Figure 2.1, an example taken from Schwartz's paper on R-Learning [17], is one environment where an agent using discounting will find a policy that is detrimental to the long-term well being of the agent. Here, the immediate rewards are in parentheses and the titles of the nodes in quotes. An agent starts on node 1, or "Earth." We see that there are two immediate paths which can be chosen by an agent on "Earth", A or B, and these paths have rewards of $+0$ and $+1000$, respectively. After the initial choice of path of A or B, the agent will remain in the same state indefinitely. In "Hell", an agent receives a reward of -1 for taking an action. In "Heaven", the reward for an action is $+1$. When concentrating on the near future, as discounting does, path B looks like the better choice. We can see, however, that if time goes on forever, the best policy is to choose path A and stay in "Heaven". This is the choice of an average reward learner.

In all finite MDP's (an MDP with a finite number of states) there is some γ close to 1 which will cause the discounted version of some reinforcement learning method to agree with the undiscounted version [17]. Though higher values of γ allow the ability to see further ahead in time (and thus visualize more long

term rewards and become closer to average reward methods), they also cause slow convergence [2, 17].

2.3. Dynamic Programming

The above section discussed different optimization criteria. In order to determine the best actions to take in each state, and thus find an optimal policy, π^*, we must move into the realm of value functions. A value function is a function that estimates the worth of being in a certain state by approximating the estimated future return of that state as a function of the state.

Dynamic Programming (DP) is used to learn value functions, and can be based on either discounted or average reward methods. It allows agents to avoid an exhaustive search of the "state-sequence space" (not to be confused with the *state*-space, which must be exhaustively searched) [2]. Classical dynamic programming requires that the problem being solved be a *known* Markov decision process. This means that the agent knows both the probability transition function representing how the environment changes with every action, and also what rewards are returned from every action taken.

In order to create an association between states and values we use Bellman Equations, which can take many forms, but we will begin with the two simplest. V^π is equal to:

$$V^\pi(s) = E_\pi[R_T|s_t = s] = r_{s,\pi(s)} + \sum_{s'} \gamma P(s'|s,a)V^\pi(s'), \tag{2.2}$$

where $V^\pi(s)$ is the value of starting from state s, and thereafter following a policy π, E_π is the expected value when following π, and $r_{s,\pi(s)}$ is the immediate reward received when starting in state s and following policy π. Expressing this in terms

of 'Q'-values, which associate values to state-action pairs, rather than just to the state [26]:

$$Q^{\pi}(s,a) = E_{\pi}[R_T|a_t = a, s_t = s] = r_{s,\pi(s)} + \sum_{s'} \gamma P(s'|s,a)V^{\pi}(s'), \tag{2.3}$$

where $Q^{\pi}(s,a)$ is the cost of taking action a from state s and from then on following policy π. DP is used to solve these Bellman equations and fill in tables representing state and state-action values, which will aid the agent in the decision of which actions to take.

If presented with multiple choices, a greedy decision will always pick the largest value possible. A policy is greedy with respect to a value function over states if it always chooses an action that maximizes the expected sum of the immediate reward received and the value of the next state. A policy is greedy with respect to a Q-value function over state-action pairs if it chooses an action that maximizes its Q-value in any state. An optimal policy is one that is greedy with respect to its value function. Optimal policies can and only optimal policies can be shown to be "greedy with respect to" their evaluation function (V). The value of the state s when following the optimal policy π^* is denoted by $V^*(s)$ and has the following Bellman equation:

$$V^*(s) = \max_a [r + \sum_{s'} \gamma P(s'|s,a)V^*(s')]. \tag{2.4}$$

The value of state s and action a picked when following the optimal policy π^* is denoted by $Q^*(s,\pi^*(s))$ and has the following Bellman equation:

$$Q^*(s,\pi^*(s)) = r + \sum_{s'} \gamma P(s'|s,a)V^*(s'). \tag{2.5}$$

Referring to Equations 2.4, and 2.5, a relation may be drawn between $V^*(s)$ and $Q^*(s,\pi^*(s))$:

$$V^*(s) = Q^*(s, \pi^*(s)) = \max_{a \in A(s)} Q^*(s, a). \tag{2.6}$$

In order to find what action is the best action to take, an agent must first evaluate the Q-values for all possible actions from the current state s by using Equation 2.7 and then find the best action by using Equation 2.6 [2].

Equations 2.2 and 2.3 are called Bellman Optimality Equations, named after Richard Bellman [20]. The values calculated by these equations are a type of 'rating' that is learned for each state. States or state-action pairs with a high rating (or value) will tend to return better rewards in the long run, and states with low ratings (or values) will tend to return worse rewards in the long run.

A policy (a mapping from states to actions) is obtained by choosing actions greedily with respect to the value function, which is learned by solving a Bellman equation. The theory behind the dynamic programming says that an optimal policy, π^*, will be obtained by always choosing to take the action of maximal value, or by using a greedy strategy, with respect to the values defined by the Bellman Equation.

2.3.1. Value Iteration

Before we continue with any discussion of various aspects of dynamic programming, we must first introduce the basic outlines of a learning algorithm. Value iteration is a method of assigning a value to each state. The update for value iteration is:

$$\begin{aligned} V_{k+1}(s) \leftarrow & \; max_a E[r_{t+1} + \gamma V_k(s_t = s, a_t = a)] \\ = & \; max_a \sum_{s'} P(s'|s, a)[r_{s,a,s'} + \gamma V_k(s'). \end{aligned}$$

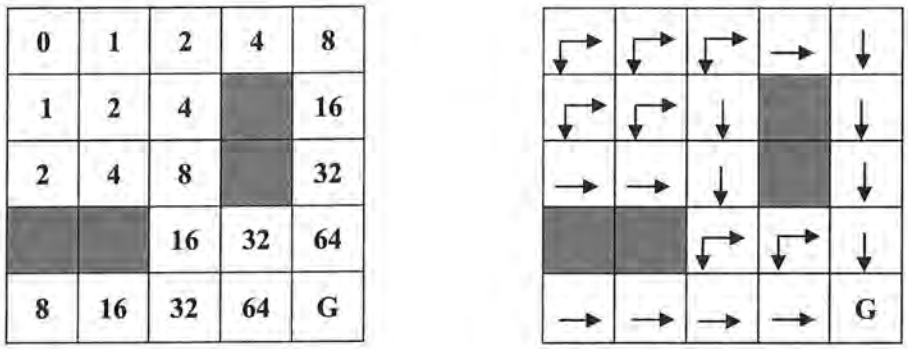


FIGURE 2.2. An example of value iteration.

For an example, take into account a simple grid environment. In this environment, the agent wants to find the path of greatest value. Figure 2.2 shows such an environment, with darkened squares representing impenetrable walls and the square labeled 'G' representing the goal state of the agent (here a state is simply the agent's location in the grid). In this environment, the default reward is 0, and a reward of +64 is given if the agent reaches the goal state. In figure 2.2, γ is set equal to 0.5. For simplicity, this environment is deterministic. Part (a) shows the values for each square (or, rather, for each state) in the grid after 7 iterations of the value iteration algorithm.

Value iteration involves a series of updates on the values of all states in the state space; during this stage all state values are determined. The values shown in figure 2.2 are the result of value iteration (they are not the final values, but they are close). For example, in the top right-most square of the grid, the best move is to go down, as this will yield a reward of +16, whereas the only other possible move yields a reward of +4. Part (b) of figure 2.2 shows the

resulting greedy policy, given that the agent uses the state values shown in part (a). Multiple arrows in one square indicate that more than one action has the same return, and so no one action is favored; either action is equally greedy at this point in execution. There are two different techniques for using value iteration to solve Bellman equations. Synchronous methods update current values using only figures from previous iterations. An example of synchronous update in the context of discounted optimization is:

$$V_{t+1}(s) \leftarrow \max_a \sum_{s'} [r_{s,a} + \gamma P(s'|s,a) V_t(s')], \tag{2.7}$$

This equation written in terms of Q-values is:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \beta[r + \gamma \max_{a' \in A(s')} Q_t(s',a') - Q_t(s,a)]. \tag{2.8}$$

In synchronous methods, the total value state space will increase and approach convergence asymptotically, even if every individual state's value does not increase [2].

In asynchronous methods, there are less strict requirements as to the manner of backing up state values. Asynchronous methods can also update new function values based upon past and present values of the states. An example of an asynchronous update in the context of discounted total reward is:

- For each $k = 0, 1, \ldots$, if $S_k \subseteq S$ is the set of states whose value is updated at iteration k, then V_{k+1} is computed as:

$$V_{k+1}(s) \leftarrow \begin{cases} \max_{a \in A(s)} [r_{s,a} + \sum_{s'} \gamma P(s'|s,a) V_k^{\pi^*}(s')] & \text{if } s \in S_k; \\ V_k(s) & \text{otherwise [2].} \end{cases} \tag{2.9}$$

This means that not all state values must be backed up in every iteration, although every state must be backed up eventually. Most strikingly, S_k can be a set consisting of only one state.

2.3.2. Policy Iteration

When the agent has a complete and accurate model of the MDP, that is, it knows the true values of the models $P(s'|s,a)$ and $r_{s,a}$, the problem may be solved off-line. This can be seen as the agent using simulations of the actual environment in order to train itself and non-adaptive control in order to solve the problem it has been issued. Online methods and adaptive control will be mentioned later.

Non-adaptive control[2] is the type of learning which we have been addressing for which DP is suitable. It requires knowledge of both the transition probability $P(s'|s,a)$ and the reward received, $r_{s,a}$, from taking action a in state s. When this information is not known, the resulting problem is called an unknown MDP or adaptive control. One needs to either learn the MDP and then solve it, which is called a model-based approach, or learn to solve the MDP directly, which is called the model-free approach[3]. Learning action models through the use of Bayesian networks is an example of using a model-based approach to adaptive control; the probabilities represented in each node are continually updated with each iteration of the program. Representing and learning the Q-values and using them for action selection is a model-free approach to adaptive control.

Policy iteration is different from the methods defined above. In the previous value iteration methods, values based on states (or state-action pairs) were explicitly stored, and a policy was something that just happened to result as a

---

[2] "Control" here refers to the determination of actions.

[3] A Markov Decision Problem with incomplete information *is not* the same as a partially observable Markov Decision Problem (POMDP), in which the whole of the *state* information is not known. The latter problem is beyond the scope of this paper.

consequence of taking actions greedily with respect to the value function. In policy iteration, there are two main sections to the basic algorithm: 1. policy evaluation, and 2. policy improvement. During the policy evaluation, the value of the current policy is determined for all starting states. This is done by solving the Bellman Equation 2.2 using the update:

$$V^{\pi}(s) \leftarrow \sum_{s'} P(s'|s, \pi(s))[r_{s,\pi(s),s'} + \gamma V^{\pi}(s')],$$

where $V^{\pi}(s)$ is the value of state s taken when starting in state s and following policy π.

During policy improvement, a new policy π' is found which is greedy with respect to V^{π}. In other words,

$$\pi'(s) = argmax_a \sum_{s'} P(s'|s, a)[r_{s,a,s'} + \gamma V^{\pi}(s').$$

If the old policy π and the new policy π' are the same in all states, then the policy has become stable, and therefore, it has theoretically reached π^*. If not, the evaluation and improvement stages are repeated until the policy *does* converge.

Policy iteration is guaranteed to reach an optimal policy in a finite number of iterations; however, a lot more computation is typically required than in value iteration. This extra computation comes from evaluating the policy every time through, perhaps searching the entire state space several times in one iteration. This is one of the reasons why this method is not quite as popular as the previously described value iteration. Another reason may lie in the fact that the theory behind improved versions of policy iteration such as optimistic policy iteration is not quite as well understood as the theory for value iteration [2].

2.4. Reinforcement Learning

Reinforcement learning involves treating an artificial agent as one would treat a little child. Initially, this child is placed inside of an environment of which it knows nothing about. The child is given rewards (positive and negative) for every action that it may make. The child then must learn what actions to choose to optimize its total return. Since the reward function and the next-state function are not known, dynamic programming methods cannot be directly used.

There are two different types of reinforcement learning methods for learning MDP's with incomplete information. Some reinforcement learning methods learn state values indirectly by constructing models of the unknown components of the environment, namely $P(s'|s,a)$ and $r_{s,a}$. These techniques are known as indirect, or model-based, methods. Another variation of reinforcement learning is to directly model the policy using values based on state-action pairs. Methods using this version do not need to create any explicit models of the domain; they apply state-action values to form policies and determine π^*. These are known as direct, or model-free, methods.

2.4.1. Discounted Reinforcement Learning

- **Adaptive Real Time Dynamic Programming - ARTDP** -

When asynchronous dynamic programming is run concurrently with the actual execution of an MDP, it is said to be running in Real Time [2]. Differences between Real Time execution and simulation mode include the fact that the control decisions of the program will be based on the latest updates of stored values. The policy to be used is usually greedy according to these latest value estimates.

At any time t only state s_t will have its value updated (this means that not all states will be backed up during every single time step, thus the asynchronous methodology).

If the agent is provided a Markovian decision problem with incomplete information (either $P(s'|s,a)$ or $r_{s,a}$ is not known), then the problem must be solved using reinforcement learning or adaptive control. In ARTDP (Adaptive Real-Time Dynamic Programming), the program uses the agents models of the MDP (the models that agent stores of $P(s'|s,a)$ and $r_{s,a}$) and acts according to the assumption that they are the true models. This is called the certainty equivalence principle in the adaptive control literature. Due to this assumption, one must keep in mind that the agent's current perception of the domain is not necessarily (in fact, at the beginning it is most likely not even remotely) close to the true MDP. When learning evaluation functions, a greedy policy will suggest the action that looks the best according to the current value function. However, it is easy in this way to get lured into a sub-optimal policy: depending on which ones are updated first, various actions will have the optimal value at different points during training. A greedy policy will always select an action which currently has the highest value. In the case of a sub-optimal action being executed more frequently at the beginning of execution, it may have a higher value than the true optimal action does. In order for the agent to locate all true components of π^*, it is imperative that the agent not get stuck in these "local maxima." The way to assure that this is done, is to guarantee exploration–that the probability an action a is chosen in state s is greater than zero for all $a \in A(s)$. A common way that this is done is to use an ϵ-greedy exploration strategy.

This first introduces the need for exploration of the environment, and the big conflict between exploration and the exploitation of the algorithm in use. In

any reinforcement learning algorithm, there needs to be *induced* and guaranteed exploration of the entire environment in which the agent finds itself. Without this exploration, the agent cannot even attempt at finding true values for every state in S. The most popular method of exploration is an "ϵ-greedy" method, which takes random actions with probability ϵ ($0 < \epsilon < 1$) and greedy actions with probability $1 - \epsilon$. One can see the conflict here. With too high of an ϵ (too much exploration) the agent will become completely random. There needs to be some element of control issued here–some measure of exploitation of the values that are currently being held. The issue of exploration will be explored more in Section 3.

• Temporal Difference Learning •

Temporal difference (TD) learning methods are a type of Reinforcement Learning first developed by Sutton [2]. TD-Learning is a type of discounted reinforcement learning which bases the update of its Bellman equation upon the difference in the estimate of values between successive time steps [20, 23]. This means that, as in other asynchronous, online methods, TD-Learning uses estimations of the value function's future worth and observed values of the received reward in order to update the current estimate of the value function. TD-Learning updates once at every time step, as do other online methods.

To put all of this more concretely, the Bellman update equation for the simplest kind of TD-Learning, TD(0), is:

$$V(s) \leftarrow V(s) + \beta[r + \gamma V(s') - V(s)]$$

The learning rate, β, is necessary to keep track of how strongly an old state value will be considered when updating a its value. TD(0) has been shown to converge with probability 1 if β is decayed appropriately [19].

TD(0) requires examining only the current reward received. More general versions of TD-Learning, n-step TD methods, base the reward used in their backup equations on the results of n steps (instead of just the immediate return, as in TD(0)). Rewards used in these methods can be thought of as a sort of truncated total discounted reward (Equation 2.1). For example, the reward used in TD(1) is:

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s').$$

where r_{t+1} is equivalent to r (the use of this different notation will be revealed in the next equation). Above, the final term is used to approximate the truncated section of the discounted total reward. The general formula for finding the reward in n-step TD-Learning is:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma V_t(s_{t+n}).$$

We can now generalize the backup equation for n-step TD-Learning as

$$V(s) \leftarrow V(s) + \beta[R_t^{(n)} - V(s)], \qquad \forall s \in S.$$

TD(λ) is a type of n-step TD-method that uses a look-ahead technique that averages future rewards together through use of the parameter λ (where $0 \leq \lambda \geq 1$). This is the theoretical equivalent of looking ahead n moves in each state visited and combining the rewards of each of the n future states with experimentally decaying the weight of the value of λ per step [20]. The formula for the reward is now changed to

$$R_t^{\lambda} = (1 - \lambda) \sum_n \lambda^{n-1} R_t^{(n)}.$$

Of course, looking into the future is not something that a reinforcement agent can do without the aid of a crystal ball, so this view of TD(λ) is purely

theoretical. When actually computing the results, TD(λ) is achieved through looking backwards in time (this somewhat varies the equations of course, but the theory remains the same).

In Tesauro's experiment of playing the game of backgammon, TD-λ was used, and the resulting agent was termed TD-Gammon [23]. In order to be able to scale their problem in large domains, TD-Gammon used multi-layer neural networks trained by the TD-λ method to approximate complex nonlinear functions[4]. TD-Gammon far out-stepped its supervised learning predecessors. In fact, the resulting TD-Gammon agent thought up playing strategies that human experts hadn't considered before, and, in some cases, it gave better results.

- **Q-Learning** -

Q-Learning is one of the most used methods in reinforcement learning. It is an adaptive, asynchronous, online, direct learning method. Like TD-Learning, Q-Learning induces exploration, usually using a random action with probability ϵ. Instead of using a function to learn values for each state, $V(s)$, a Q function is kept, which directly learns the value of state-action pairs. In other words, Q-Learning maintains a relationship of: $Q : S \times A \rightarrow \Re$.

To get the common version of the Q-Learning Bellman equation, begin by taking $\max_{a' \in A(s')}$ on both sides of Equation 2.3. This yields:

$$V^*(s) = \max_{a \in A(s)} Q^*(s,a) = \max_{a' \in A(s')} Q(s',a'), \qquad \forall s \in S.$$

where we are estimating $\max_{a' \in A(s')} Q(s',a')$, the Q-value from taking the optimal action from s', to be the value of the state. From here, we can move to the Bellman update equation for Q-Learning:

---

[4]The topic of function approximation will be discussed in Section 4.

$$Q(s,a) \leftarrow Q(s,a) + \beta[r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a)].$$

The advantage of Q-Learning is that it does not need to know or estimate the action models either to do the updates of the Q-function or select the greedy action in a given state.

2.4.2. Average Reward Reinforcement Learning

Section 2.4.1 was about discounted reinforcement learning. However, for the sake of convenience, even dynamic programming formulae previous to that section used discounting as well. We will now introduce undiscounted dynamic programming formulae that incorporate the ideas of average reward.

An average-reward reinforcement learning value function maintains the relative value of executing action a in state s under policy π. This is known as a *relative* value because we are concerned with the value of the state *in comparison to* the expected total reward of the policy on the average. This is computed by subtracting the average reward ρ^π of the policy π from the immediate reward at each step, accumulating the results, and taking its expected value. In other words, the expected value of a state might be represented as:

$$V^\pi(s) = \sum_{k=1}^{\infty} E_\pi[r_{t+k} - \rho^\pi | s_t = s],$$

or as:

$$Q^\pi(s,a) = \sum_{k=1}^{\infty} E_\pi[r_{t+k} - \rho^\pi | s_t = s, a_t = a],$$

where ρ^π is the average reward when executing policy π and has the expected value of:

$$\rho^\pi = \lim_{n \to \infty} \frac{1}{n} \sum_{t=1}^{n} E_\pi[r_t]. \tag{2.10}$$

The value V^π of policy π satisfies the following Bellman equation:

$$V^\pi(s) = r_{s,\pi(s)} - \rho^\pi + \sum_{s'} P(s'|s,a)V^\pi(s'). \tag{2.11}$$

Similarly, the value Q^π of policy π satisfies:

$$Q^\pi(s,a) = r_{s,a} - \rho^\pi + \sum_{s'} P(s'|s,a)Q^\pi(s',\pi(s')). \tag{2.12}$$

Finally, the values of the optimal average-reward policies satisfy the following equations, where ρ^* is the average reward of the optimal policy.

$$V^*(s) = \max_a [r_{s,a} - \rho^* + \sum_{s'} P(s'|s,a)V^*(s')]. \tag{2.13}$$

$$Q^*(s,a) = r_{s,a} - \rho^\pi + \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a'). \tag{2.14}$$

In the following, we describe model-free R-learning and model-based H-learning, which learn these action-based and state-based value functions for average-reward optimization.

- **R-Learning** -

Schwartz first developed the method of R-Learning in 1993 [17]. He noted that discounting considers only immediate (and possibly mediocre) rewards while ignoring future (and possibly superior) rewards. He also observed that in most reinforcement learning plots, the average reward is diagramed, but, in the case of discounted learning, this is not the method by which the agent was trained.

R-Learning (figure 2.3) is an average reward variation on Q-Learning. Only a few steps in the Q-Learning algorithm need to be adjusted in order to yield the R-Learning algorithm, and most all of them follow directly from the value function for average-reward reinforcement learning: average-adjust the rewards by

subtracting ρ (the *gain*, or average reward) from the Bellman equation, eliminate γ by letting it go to 1, and introduce an update equation for the upkeep of ρ (which is updated only when optimal actions are taken, to keep ρ as close to the true value as possible). Notation is also modified, changing Q-values to R-values. The update for R-Learning is:

$$R(s,a) \leftarrow R(s,a) + \beta[r + H(s') - R(s,a) - \rho], \tag{2.15}$$

where

$$H(s') = \max_{a' \in A(s')} R(s',a').$$

Initialize ρ and $R(s,a)$, for all s, a, arbitrarily.

Repeat Forever:

1. $s \leftarrow$ *current state*
2. Choose an action from state s by comparing $R(s,a)$ and using the ϵ-greedy exploration strategy. Let a be the action taken, s' be the resulting state, and r be the immediate reward received.
3. Take action a, observe r, and s'
4. $R(s,a) \leftarrow R(s,a) + \beta[r + H(s') - R(s,a) - \rho]$,
   where $H(s') = \max_{a' \in A(s')} R(s',a')$.
5. If $R(s,a) = \max_{a \in A(s)} R(s,a)$, then
   $\rho \leftarrow \rho + \alpha[r + H(s') - R(s,a) - \rho]$
6. $\alpha \leftarrow \frac{\alpha}{\alpha+1}$

FIGURE 2.3. The R-Learning Algorithm.

R-learning has not been proven to converge to the optimal policy [17, 11], but has been experimentally shown to reach the optimal policy, eventually, on most every problem posed to it so far.

• **H-Learning** •

H-Learning is the model-based counter-part of R-learning, and the average-reward version of ARTDP [21]. In H-Learning (figure 2.4) an ϵ-greedy search strategy is again used. The models that are learned are: the average immediate reward received in state s from taking action a (this table is referred to as $r_{s,a}$), the probability of moving to state s' from state s when taking action a (this table is referred to as P and takes care of stochasticity in the environment). H-Learning was found to perform either as well as or better than ARTDP in a number of domains [21]. The update for H-Learning is:

$$h(s) \leftarrow \max_{a \in A(s)} \{r_{s,a} + \sum_{j=1}^{n} P(j|s,a)h(j)\} - \rho. \tag{2.16}$$

Upon inspection of the algorithms presented in figures 2.3 and 2.4, one can see how similar the R- and H-Learning processes are. Besides the difference in modeling, the biggest changes between the two methods are the order of the updates of the value function and the gain, and the update equations themselves; the latter of which is the most prominent distinction. Despite their disparate appearance, these two updates (Equations 2.15 and 2.16) are actually quite similar.

First, let's break down the update for H-Learning (Equation 2.16). We see that it begins by finding the action a that will maximize the sum of the modeled immediate reward and the weighted average of the h-value of all possible next states j. From all this, ρ is subtracted. Turning now to the update for R-Learning (Equation 2.15), we see that, in the short term, it adds together the max over all next actions a' of the value $R(s', a')$ (this whole term may be, and is, referred to as $H(s')$ because of its congruency with H-learning's update) and the immediate reward, then subtracts ρ. To compare the two updates, it becomes necessary to think in a larger scale, time-wise.

Initialize.

Repeat Forever:

1. $s \leftarrow$ *current state*
2. Choose an action from state s by comparing the h-value of the resulting state and using ϵ-greedy exploration strategy. Let a be the action taken, s' be the resulting state, and r be the immediate reward received.
3. Update $P(s'|s,a)$.
4. Update $r_{s,a}$.
5. *GreedyActions(s)* $\leftarrow$ All actions $a \in A(s)$ that maximize $r_{s,a} + \sum_{j=1}^{n} P(j|s,a)h(j)$
6. If $a \in GreedyActions(s)$, then
   (a) $\rho \leftarrow \rho + \alpha(r_{s,a} + h(s') - h(s) - \rho)$
   (b) $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
7. $h(s) \leftarrow \max_{a \in A(s)}\{r_{s,a} + \sum_{j=1}^{n} P(j|s,a)h(j)\} - \rho$
8. $s \leftarrow s'$

FIGURE 2.4. The H-Learning Algorithm.

After a long period of time has passed, all state-action combinations will have been attempted and updated. It goes almost without saying that the only way in which an agent using R-Learning will see a state s' in its update is if the action a moves the agent into this state. Action a is only going to move from state s into state s' with a probability of $P(s'|s,a)$, and thus this is the percentage of the time that the update for value $R(s,a)$ will involve using the numbers for the state s'. Therefore, one might call this a sort of weighted average that takes place over a longer period of time. This is also exactly the same idea behind the update equation for Q-Learning.

2.5. Hierarchical Methods

Probably the largest downfall of all reinforcement learning methods are the extremely massive state spaces required, which necessitate meticulous exploration in order for the agent being trained to learn anything. Hierarchical methods strive to counteract this downfall by incorporating methods of abstraction to break down the entire MDP being solved into a hierarchy of simpler "sub-MDPs" (also called subtasks or subroutines). Sub-MDPs require less knowledge than the original "root" MDP (which encompasses the overall goal of the agent). Irrelevant features[5] in the state representation can be eliminated, depending on the current subtask (a feature is irrelevant for a subtask if it doesn't affect the next state or the value of the state for that subtask).

In hierarchical reinforcement learning, each subtask finds the policy that is optimal within the subtask. The overall policy which is chosen is a combination of all of the policies of the sub-MDPs. The goal of the overall program is to find a *recursively optimal* policy. Dietterich [7] defines recursively optimal as "an assignment of policies to each individual subtask such that the policy for each subtask is optimal given the policies assigned to all of its descendents." Recursively optimal is not, however, guaranteed to be optimal overall, or optimal policy over all hierarchical policies.

Hierarchical Semi-Markov Q-Learning and Dietterich's MAXQ [6, 7] algorithms operate on the above principles to find recursively optimal policies. Parr and Russell's hierarchical abstract machines learn by finding *hierarchically optimal* policies [13]. A hierarchically optimal policy is one that is optimal when given

---

[5] *Features* are various important elements of a state

the restraints of hierarchies. This means that not all of the subtasks' policies are necessarily optimal, and therefore, hierarchically optimal is not necessarily recursively optimal and vice versa.

Hierarchical reinforcement learning has been applied, and, in most cases, found to outperform its nonhierarchical counterparts [6, 18]. A hierarchical version of H-Learning (HH-Learning) is used by Seri and Tadepalli [18]. In their investigation, they found HH-Learning to improve upon H-Learning in an automatic guided vehicle (AGV) scheduling task.

3. AUTO-EXPLORATORY REINFORCEMENT LEARNING

Most of the algorithms described above involve enforcing exploration by inserting random actions with some probability. The major problem with this ϵ-greedy search strategy is that it forces the agent to choose sub-optimal actions in order to efficiently explore the state space. It would be much preferred to always take greedy actions. Some advantages to always taking greedy actions are: removing sub-optimal reward returns, getting rid of the parameter ϵ (and all the adjustment that it may require), and abstaining from blind searching. This is what an auto-exploratory search strategy attempts to do: thoroughly search the available state space, and yet always take greedy moves.

The main theory behind an auto-exploratory search comes from noting that the gain, ρ (approximately the average reward), is subtracted from the R-value update in the Bellman equation of R-learning [12]:

$$R(s,a) \leftarrow R(s,a) + \beta[r + H(s') - \rho - R(s,a)].$$

Since the value function in average reward reinforcement learning, R in this case, is measured *in relation to* ρ, for larger values of ρ the total value of $R(s,a)$ is decreased. The auto-exploratory search strategy utilizes this and keeps ρ at a high value.

To see an example of this, direct your attention to figure 3.1. This is a grid world environment similar to that shown in figure 2.2, which demonstrates value iteration. Figure 3.1 is different than figure 2.2 in that it lists all possible actions in every square (R = right, U = up, D = down, L = left), and includes the current value of that state (the state being, again, the square in the grid) action pair. Moves that cannot be made are denoted by '–'. The goal state is the bottom-left-hand corner square and the environment is deterministic. The reward

R = -0.5 U = -- D = -0.75 L = --	R = -0.5 U = -- D = -0.5 L = -0.5	R = -- U = -- D = -0.5 L = -0.5
R = -0.5 U = -0.5 D = -0.5 L = --	R = -0.5 U = -0.5 D = -0.5 L = -0.5	R = -- U = -0.5 D = 0 L = -0.5
R = 0 U = -0.5 D = -- L = --	R = 0 U = 0 D = -- L = 0	GOAL

(a)

R = 0 U = -- D = 0 L = --	R = 0 U = -- D = 0 L = 0	R = -- U = -- D = 0 L = 0
R = 0 U = 0 D = 0 L = --	R = 0 U = 0 D = 0 L = 0	R = -- U = 0 D = 0 L = 0
R = 0 U = 0 D = -- L = --	R = 0 U = 0 D = -- L = 0	GOAL

(b)

FIGURE 3.1. An example of R-values in a grid-world environment.

for reaching the goal is +5, the default reward is +0, $\beta = 0.5$ (β is a learning parameter that tells how seriously to take the new value), and all state-action values are initialized to 0. A greedy search strategy is used, with ties broken randomly. In part a, ρ is held at 1, in part b it is held at 0. Both a and b have been through 19 iterations. Notice how in part a, where ρ is set to 1, all visited states-action pairs are less than 0. It is obvious where the agent has been and where it has not been. In part b, where ρ is set to 0, all state-action values are still 0, and thus after 19 iterations, the agent is still blind as a bat. The *larger value of ρ seems to aid in the exploration* of the state-action space: Already visited states and actions will have a smaller value than the more enticing unexplored regions, thus forcing the agent to try unfamiliar zones while always taking greedy actions. This ideology is known more simply as optimism under uncertainty [18].

We must now find some way in which to keep the value of ρ "sufficiently" large. This is done by introducing new parameters ρ_{min} and ρ_{max}, which the value of ρ is forced to remain between. ρ_{max} is set at some unattainably high value preferably just above ρ^{π^*} and ρ_{min} is set close to, but lower than, the value of the policy which the agent is expected to learn (ρ_{min} is a parameter which needs to be tuned according to the environment and other parameters being used). The value of α, the learning rate of ρ, is decayed so as to reduce the rate of exploration as more states and actions are being evaluated. If the current policy being used is sub-optimal, then ρ, which converges to its average reward, will eventually reach ρ_{min}, in which case ρ is immediately bumped back up to ρ_{max} (and its learning rate, α, can also be bumped up to the initial value (α_0) with it), forcing the agent to explore new regions. Hence, we have:

If $\rho < \rho_{min}$, then
$\rho = \rho_{max}$ and $\alpha = \alpha_0$

Initialize.

Repeat Forever:

1. $s \leftarrow$ *current state*
2. Take an action $a \in A(s)$ that maximizes $R(s, a)$ in the current state s. Let s' be the resulting state, and r be the immediate reward received.
3. $R(s, a) \leftarrow R(s, a) + \beta(r + H(s') - \rho - R(s, a))$
4. $\rho \leftarrow \rho + \alpha(r - \rho)$
5. $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
6. If $\rho < \rho_{min}$,
   $\rho = \rho_{max}$ and $\alpha = \alpha_0$
7. $s \leftarrow s'$

FIGURE 3.2. The AR-Learning Algorithm.

Auto-exploratory R-Learning (AR-Learning) is the auto-exploratory version of Schwartz's R-Learning (see figure 2.3). Note in this algorithm the definition of

$$H(s') = \max_{a' \in A(s')} R(s', a') \tag{3.1}$$

is inspired from the H-Learning algorithm (figure 2.4). This is the algorithm that we have based our research on.

Initialize.

Repeat Forever:

1. $s \leftarrow$ *current state*
2. Take an action $a \in A(s)$ that maximizes $R(s, a)$ in the current state s. Let s' be the resulting state, and r be the immediate reward received.
3. Update model $P(s'|s, a)$.
4. Update model $r_{s,s',a}$.
5. $R(s, a) \leftarrow \sum_{j=1}^{n} [P(j|s, a)(H(j) + r_{s,j,a})] - \rho$
   where $H(j) = \max_{a' \in A(j)} R(j, a')$
6. $\rho \leftarrow \rho + \alpha(r_{s,s',a} - \rho)$.
7. $\alpha \leftarrow \frac{\alpha}{\alpha+1}$.
8. If $\rho < \rho_{min}$,
   $\rho = \rho_{max}$ and $\alpha = \alpha_0$.
9. $s \leftarrow s'$.

FIGURE 3.3. The AH-Learning Algorithm.

Auto-exploratory H-Learning (AH-Learning) is the auto-exploratory version of H-Learning [21]. AH-Learning (figure 3.3) differs from AR-Learning only in its Bellman equation and the fact that it is model-based, learning models for rewards and for probability.

4. FUNCTION APPROXIMATION

For the methods and update functions mentioned above, we have assumed a look-up table representation. That is, for each value (V, Q, H, R, or otherwise) there is a one-to-one mapping to some matrix that stores all values. Such a representation can always be achieved when the number of states is finite. In very large state spaces, however, we come across two major problems: 1. The physical space required to hold the value table grows extremely large, and 2. The time needed to adequately explore the entire state space and come up with an optimal policy expands to unacceptable proportions.

It is convenient to represent each state s as a feature vector $\vec{\phi}_s$. A feature is some important aspect of the state that needs to be noted. For example, in a chess environment one feature might be where my king is currently located on the board. Function approximation seeks to learn a set of weights, or parameters, $\vec{\theta}$, that will, when combined with a set of corresponding features $\vec{\phi}_s$, yield an approximation for a value function. For example, in linear function approximation the parameters and the feature values are combines in a linear fashion:

$$V_t(s) \approx \sum_{i=0}^{n} \theta_t(i)\phi_s(i) = \vec{\theta}_t \cdot \vec{\phi}_s, \tag{4.1}$$

where $\vec{\phi}_s$ is the feature vector of state s and $\vec{\theta}_t$ is a vector of adjustable coefficients at time step t. The learning of $\vec{\theta}$ can be done in several ways. We will present two different versions on the method of gradient descent.

Before we move on it is important to note that, though function approximation is good for speeding up the learning process and decreasing necessary space requirements in large state spaces, it also has its drawbacks. While shown to work in some circumstances [19], the errors introduced by approximation are

known to occasionally lead to sub-optimal policy convergence [17, 24] and even divergence [1].

4.1. Gradient Descent Function Approximation

It is our objective to ensure that the values to which $\vec{\theta}$ is set will make the agent take actions which result in maximizing expected returns in the long-term. The way in which we will do this is through the use of TD-error, or temporal difference error.

TD-error is based on a comparison of state values: how much better/worse off is the value of the current state of the agent compared to what it should be based on the next state's value? The equation of the TD-error, δ, for a discounted algorithm is:

$$\delta = r + \gamma V(s') - V(s).$$

Since we are going to be dealing with average-reward reinforcement learning, let's put this in the terms of an average-reward method (more specifically, R-Learning),

$$\delta = r + H(s') - \rho - R(s, a).$$

Now shift this all into the terminology of function approximation,

$$\delta = r + H(s') - \rho - \vec{\theta_t} \cdot \vec{\phi_s}. \tag{4.2}$$

After solving for the TD-error, an appraisal of δ will reveal the usefulness of taking action a. Our goal is to get the error to be as small as possible. If the given value for the next reward $(r + H(s') - \rho)$ is greater than the current estimate of the R-value, $\vec{\theta_t} \cdot \vec{\phi_s}$, then δ will be positive and R-value will be adjusted upwards by adjusting $\vec{\theta_t}$. If the δ is negative, then the action is considered one that might not be such a good idea in the future.

4.1.1. Linear Function Approximation

The method of gradient descent involves finding a minimum in a function by moving in a "downward" direction, as pointed to by the gradient (remember that the gradient is a *vector* operation, and maintains directionality after it is performed: we simply need to move in the direction that the gradient is negative to decrease the value of the function). We want to find the place where the TD-error is the least possible value. In order to do this, we need to critique the error in reference to the only entity that we can change, the parameter $\vec{\theta_t}$. To facilitate this, we take the partial derivative with respect to $\vec{\theta_t}$ (or, in other words, find the component of the gradient dependent upon $\vec{\theta_t}$).

To begin, we need to come up with an error measure that will, after the derivative is taken, leave things in terms of $\vec{\theta_t}$. Most people use the mean squared TD-error (MSE):

$$MSE = \frac{1}{2}(r + H(s') - \rho - \vec{\theta_t} \cdot \vec{\phi_s})^2.$$

The next step is to take the partial derivative of the MSE with respect to $\vec{\theta_t}$:

$$\frac{\partial(MSE)}{\partial\vec{\theta_t}} = \frac{\partial}{\partial\vec{\theta_t}}[\frac{1}{2}(r + H(s') - \rho - \vec{\theta_t} \cdot \vec{\phi_s})^2].$$

Through the application of the chain rule we find

$$\frac{\partial(MSE)}{\partial\vec{\theta_t}} = (r + H(s') - \rho - \vec{\theta_t} \cdot \vec{\phi_s})\frac{\partial}{\partial\vec{\theta_t}}(r + H(s') - \rho - \vec{\theta_t} \cdot \vec{\phi_s}).$$

Taking the final partial derivative to finish yields

$$\nabla_{\vec{\theta_t}} MSE = (r + H(s') - \rho - \vec{\theta_t} \cdot \vec{\phi_s})(-\vec{\phi_s}). \tag{4.3}$$

To shift the parameter vector a small amount in a decreasing direction, we subtract a small fraction of the gradient to form an update for $\vec{\theta}$ [27, 5]:

$$\vec{\theta_{t+1}} \leftarrow \vec{\theta_t} - \beta \nabla_{\vec{\theta_t}}(MSE).$$

Plugging in Equation 4.3, this becomes

$$\vec{\theta_{t+1}} \leftarrow \vec{\theta_t} + \beta(r + H(s') - \rho - \vec{\theta_t} \cdot \vec{\phi_s})\vec{\phi_s}. \tag{4.4}$$

4.1.2. Neural Networks

This section is for completeness of different function approximation methods, and may be skipped without interrupting the flow of the rest of this paper.

Neural networks are widely used for function approximation in reinforcement learning. A neural network is a weighted graph composed of a group of several nodes, or units, connected together by links, and organized into layers. In this paper we will talk about feed-forward networks, which require all links to be unidirectional and the network to be a directed acyclic graph[1]. A unit, one node in a neural network, can be one of three different types of units supported by a neural net. Input units receive information directly from the environment. Output units yield the final computations of the entire network. Hidden units reside in-between the input and output units. Hidden units, unlike the two other types, are not a required component of neural networks; that is, it is possible for neural networks to exist that do not contain any hidden units at all.

As mentioned above, units are arranged into separate layers (figure 4.1). The bottom layer of the network contains the only input units (and nothing else).

---

[1]There *are* types of neural nets which do not follow these requirements and have cycles and/or bi-directional links. For simplicity's sake, however, we will not go into them here.

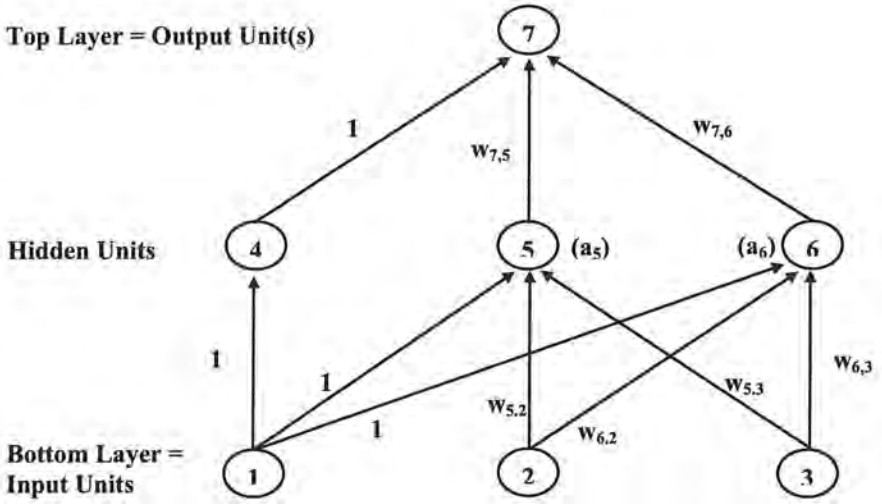


FIGURE 4.1. A simple neural network, organized into three layers.

This layer is followed by a variable number of layers containing hidden units. At the top of the network we find the top layer, which contains all of the output units in the network. Here we can appreciate the name given to hidden units fully: the outside agent, who gives input to one side of the network and receives output from the other, does not directly see these dividing layers. Thus, the in-between units are "hidden" from the user of the neural network.

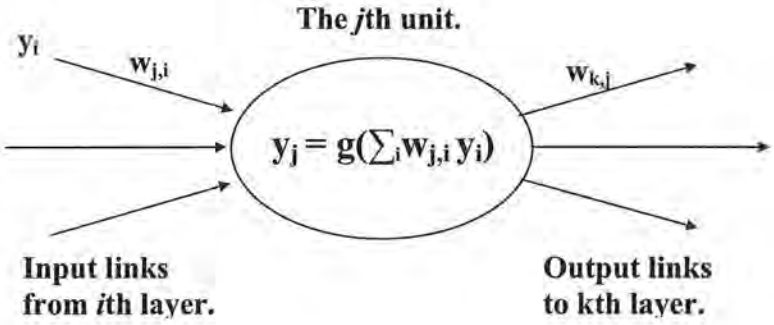


FIGURE 4.2. One unit in a neural network (adapted for this paper from Russell and Norvig).

Each unit j (figure 4.2) receives input links from units in the previous layer (except, of course, in the case of input units, which receive their input directly from the environment). Almost all networks have special inputs called biases. Biases are extra inputs that always have a value of 1, and similar to the θ_0 term of linear function approximation, allow for the existence of a constant term in the value function's approximation. All units also have output links to units in the next layer (except, of course, for the output units, whose output is given to the agent for use in the MDP being solved). Units also have a current activation level, y_j. The activation level of a unit is computed through a nonlinear activation function, g, which is usually the same for every unit in the network. Commonly used functions are the sigmoid, sine, or step functions, which yield either a 0 or 1 result[2], changing at a threshold[3]. This threshold can be adjusted using the weights found on each link.

Calculation of the activation function for the jth unit begins by taking a weighted sum (x_j) compiled from the weights on links and activation levels of the unit from which the links emanate: $x_j = \sum_i w_{j,i} y_i = \vec{W}_j \cdot \vec{Y}_j$, where $\vec{Y}_j$ is notation for a vector containing all of the activation levels of the units in the layer i directly previous to unit j and are linked that unit, and where the weight vector, $\vec{W}_j$, corresponds to the parameter vector $\vec{\theta}$. All that is left to be done to calculate the activation level of unit j is to apply the activation function to the input, or: $y_j \leftarrow g(x_j) = g(\vec{W}_j \cdot \vec{Y}_j)$. Theoretically, the activation levels of units in

---

[2]In the case of a sine function, of course, a -1 or 1 result is obtained.

[3]In biological neurons, this would correspond to a neuron firing or not firing given a certain input.

the same layer are set in parallel, and each layer is set sequentially, starting with the bottommost [15].

The heart of a neural network lies in the weights of the links between units. The objective is to determine weights that will yield the best output given any input to the network. The most popular method of finding appropriate weights for a neural network is by using back-propagation, which implements the gradient descent approach.

The back-propagation method minimizes the error between the goal output and the actual output of the network by continually adjusting the network's weights [15, 5]. Two sweeps of calculations through the network are necessary for this to occur: one forward sweep, calculating the activation levels of all the units as described above, and one backward sweep, using the back-propagation method. The basic idea behind back-propagation is to take the gradient of the TD-error. The weights are adjusted in the opposite direction of the error gradient by sifting all of the way down through the network and figuring out each individual unit's contribution to the error along the way. At the end of this procedure, when the bottom of the network is reached, all of the weights are adjusted by some $\triangle w$. This is repeated many times until the error and the weights converge.

An example of neural networks used in reinforcement learning is Tesauros TD-Gammon [23]. In this environment, the input to the network was a representation of the current backgammon board and the output of the network was a value for that board configuration [20].

5. THE PRODUCT DELIVERY ENVIRONMENT

The environment that we used for our research was that of trucks delivering products to shops. In this environment, there are a given number of trucks, and a given number of shops, located on a grid of ten nodes (figure 5.1). The main players in this domain are the trucks, which stock the shops; the shops, which hold stock and sell it to customers; and the customers, who buy the stock from the shops. In every shop there is some probability per unit time that customers will come in and "buy stock." This occurrence is represented by the inventory of the shop being decremented by one level.

There are two rewards given in the product delivery domain: a small negative reward when any truck moves along an edge in the grid (money for gas), and a large negative reward for when a shop runs out of stock and a customer comes into the shop looking to buy (for the money that might have been earned if the shop had been properly equipped). In all other instances, no reward is given.

Trucks move along edges in the graph, and when they are located at the "truck depot" (node 0), they are refilled with stock. All trucks have the following possible actions: do nothing, move up, move down, move left, move right, or unload a certain amount of stock into a shop (the last of which is only valid if the truck is located at a shop). The moving penalty for a truck is -0.1. The penalty given when a shop runs out of inventory and a customer wants to buy something is -20.

A state's features in this environment are: inventories of all shops, loads being carried by each truck, and locations of each truck. Actions involve ordering each truck to take one of the moves mentioned in the previous paragraph. A truck does not depend upon the states of the other trucks, in any way, to move.

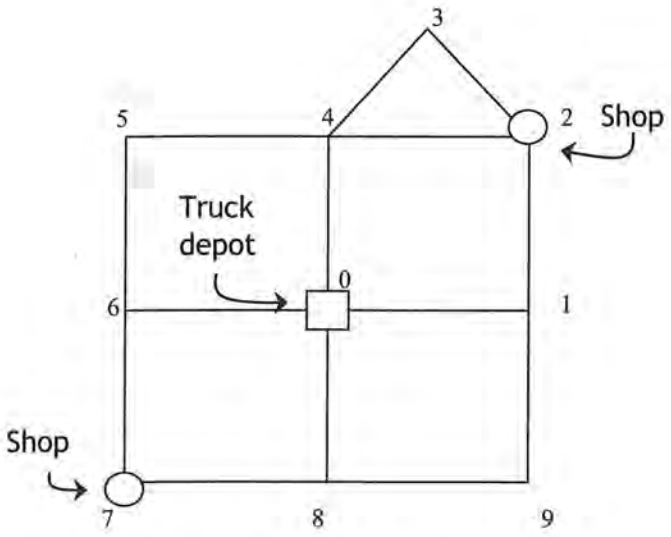


FIGURE 5.1. Grid used for the product delivery environment. Positions and number of shops can be altered.

Two trucks can inhabit the same location of the grid at the same time. The only requirements for the trucks' movements are: they can only move in a direction where a path is located, they cannot unload a greater amount than they have, and they cannot unload at all if they are not located at a shop.

The goal of this environment is to maximize the average reward received.

6. APPLICATION OF FUNCTION APPROXIMATION

For any problem worth solving, function approximation will eventually be necessary, as problems solvable by only table-based methods are either very small (and, thus, tend to be rather unexciting) or take a ridiculous amount of time to run. For instance, if we take the environment described above, and let the number of possible shop inventories/truck load levels be l, the number of shops be z, the number of trucks be τ, and the number of positions that one truck can be in be p, then, when run in a grid of 10 nodes, with 2 trucks, 2 shops, and 5 different possible inventory levels (full, $\frac{3}{4}$ full, $\frac{1}{2}$ full, $\frac{1}{4}$ full, and empty), this will have a total of:

$$l^z l^\tau p^\tau = 5^2 5^2 10^2 = 62,500 \tag{6.1}$$

states. We can see that the number of states is exponential in both the number of shops and the number of trucks. By increasing the number of shops to 5, the number of states is increased from 62,500 to 7,812,500. This is a good illustration of the motives for function approximation–somehow finding a method to reduce the number of learnable parameters and approximate the value function to help the program to be more feasible in large environments, and to run faster.

In our environment, the state features can be divided up into two categories, *linear* and *non-linear*. Here, as well as in [22, 14] we considered the inventory of the shops and loads of the trucks as linear, and the positions of the trucks upon the grid as nonlinear. Inventories are approximated as linear because of the assumption that the value of an action increases and decreases proportional to the amount of inventory stocked in a shop. A similar assumption is made for the loads of the trucks. However, there is no reason to assume such a relation exists for the locations of the trucks, thus they are denoted as nonlinear. All of

this leads to the partitioning of the value function into a set of linear equations: where there is a linear function for each combination of nonlinear feature values.

The weights, $\vec{\theta}$, have dimensions of nonlinear features (the locations of trucks) and actions. For each location vector of all trucks $\vec{l}$ and the action vector $\vec{a}$, let $\theta_{l,a}$ represent the vector of weights for the linear parameters. $\vec{\phi_s}$ is the linear feature vector representing shop inventories and truck loads. For every $\vec{\theta_{l,a}}$ and $\vec{\phi_s}$ there are the same number of components, in order that the value function may be approximated by a sum of functions of linear and nonlinear features: $\sum_{i=0}^{n} \theta_{\vec{l},\vec{a}}(i)\phi_s(i)$ (where $\phi_s(0) = 1$ in order to include a bias term, $\theta_{\vec{l},\vec{a}}(0)$).

This use of function approximation greatly reduces the number of calculations necessary. For instance, instead of searching 7,812,500 states as previously mentioned for the table-based example, the same example using this representation would involve searching 100 different truck positions (10 positions in each truck, with 2 trucks) times 7 different features, which will yield a total of only 700 different numbers to be kept to approximate states.

In order to find the update equation for this version of gradient descent for AR-Learning, note that the update equations of R-Learning and AR-Learning are the same. This means that the update for $\vec{\theta}$ in AR-Learning is the same as the one used in R-Learning, shown in Equation 4.4, except that we do this update to the linear function that corresponds to the location features of the trucks in the current state.

To find the $\vec{\theta}$-update for AH-Learning, begin with noting that the TD-error for AH-Learning is

$$r + \sum_{j=1}^{n} [P(s'|s,a)H(s')] - \rho - H(s).$$

Following the process found in Section 4.1.1, find the derivative of the mean squared error with respect to $\vec{\theta_t}$:

$$\nabla_{\vec{\theta_t}} MSE = (r + \sum_{j=1}^{n} [P(s'|s,a)H(s')] - \rho - \vec{\theta_t} \cdot \vec{\phi_s})(-\vec{\phi_s}),$$

and move this into the gradient descent update for the parameter $\vec{\theta}$

$$\vec{\theta_{t+1}} \leftarrow \vec{\theta_t} + \beta(r + \sum_{j=1}^{n} [P(s'|s,a)H(s')] - \rho - \vec{\theta_t} \cdot \vec{\phi_s})\vec{\phi_s}.$$

7. RESULTS

The purpose of this study was to investigate the strange behavior that AR-Learning was found to exhibit in [22]. In Tang's study, both table-based and function-approximated versions of H-Learning were compared to table-based versions of R-Learning, AR-Learning, and AH-Learning. In his study, both R-Learning and AR-Learning never found policies as good as those found by H and AH-Learning. In our study, AH-Learning was compared to AR-Learning. Our goal was to get AR-Learning to perform comparably to AH-Learning.

7.1. Table-based

AR-Learning was found to perform just as well as AH-Learning in table-based domains, and was observed to ultimately learn the optimal policy for the largest environment in which it was tried.

Figure 7.1 shows the results of both table-based and function approximated versions of AH and AR-Learning for an environment of 2 trucks, 2 shops. All plots shown in this section plot time on the x-axis and the average reward per time step on the y-axis, and are the result of an average of 10 runs. Here the trucks are only allowed 2 different load levels, and the same is true of the shop inventory levels. AH and AR-Learning achieve approximately the same average reward in the same number of iterations.

7.2. Function Approximation

Note the function-approximated versions of AR- and AH-Learning in Figure 7.1. The average reward found by both algorithms is lower than the average

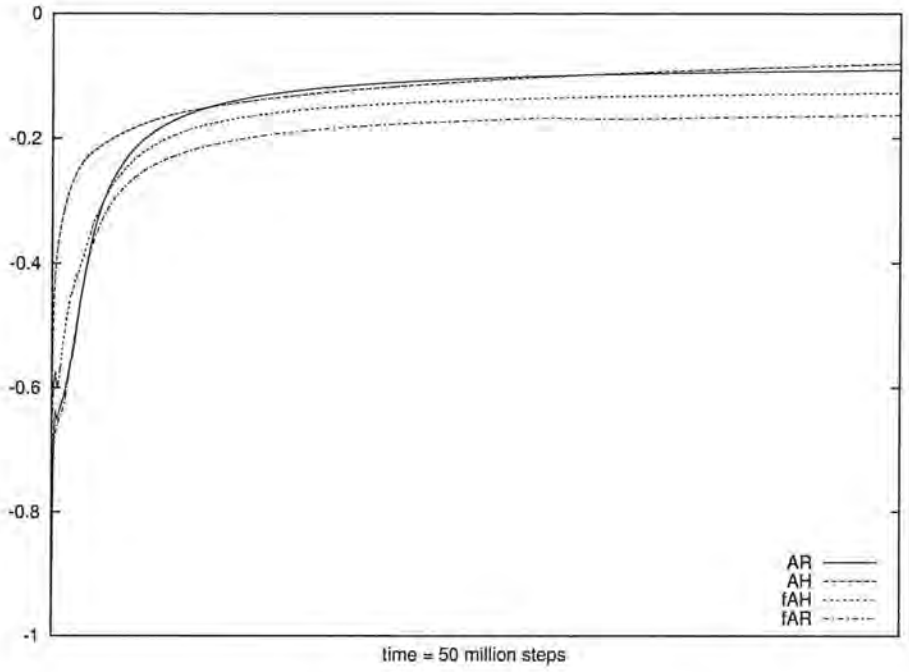


FIGURE 7.1. Versions of AH and AR-Learning in the product delivery environment for 2 trucks, 2 shops, and binary inventory levels.

reward found by the table-based methods, with AR-Learning noticeably so. Our hypothesis for the cause of this is the reliance of our piecewise linear function approximation on the values of the state features. In Figure 7.1, all state features have only two possible values. We bring up the idea of the simplest linear threshold unit, which tries to fit a linear representation to approximate the value function [5]. Perhaps the true value function does not take the form of this representation. However, when it is possible for state features take on more values, the piecewise linear approximation allows for a representation that can simulate nonlinear functions in order to represent the value function. This allows for the ability to approximate more equations with better accuracy; an ability that is

refined with the number of features. In Figure 7.2 (notice that, in this figure and the next, part (a) has a range for its x-axis of 50 million time steps, and part (b) has a range for its x-axis of 100 million time steps), features have 3 possible values each, and in Figure 7.3 features have 5 possible values each. Notice how the average rewards of the function approximated methods increase with the number of state features.

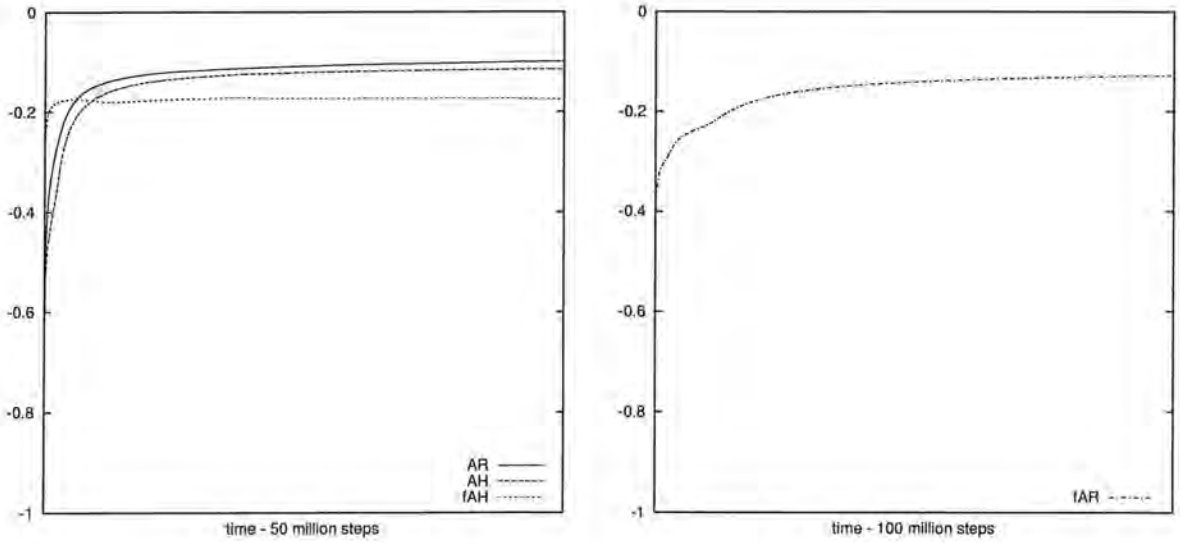
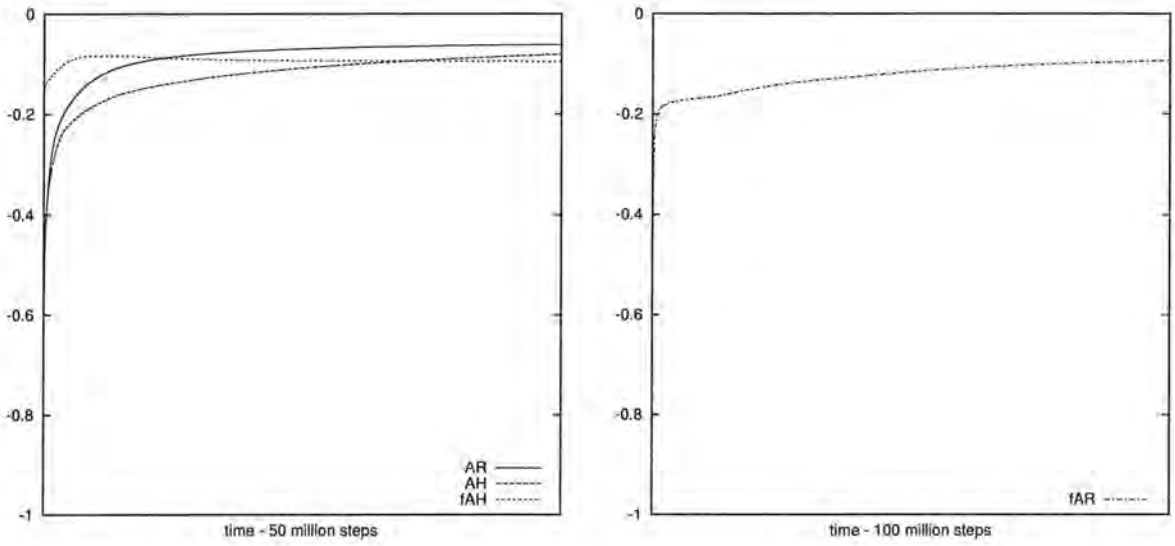


FIGURE 7.2. AH and AR-Learning in the product delivery environment with 2 trucks, 2 shops, and 3 different different inventory levels.

In domains with higher numbers of possible values for state features, function-approximated versions of both AR- and AH-Learning learn policies only slightly lower than the policy obtained by the table-based versions mentioned above, which is approximated as optimal (shown in figure 7.3). This is reasonable because the use of any function-approximation method will alter the resulting values from the non-approximated version.

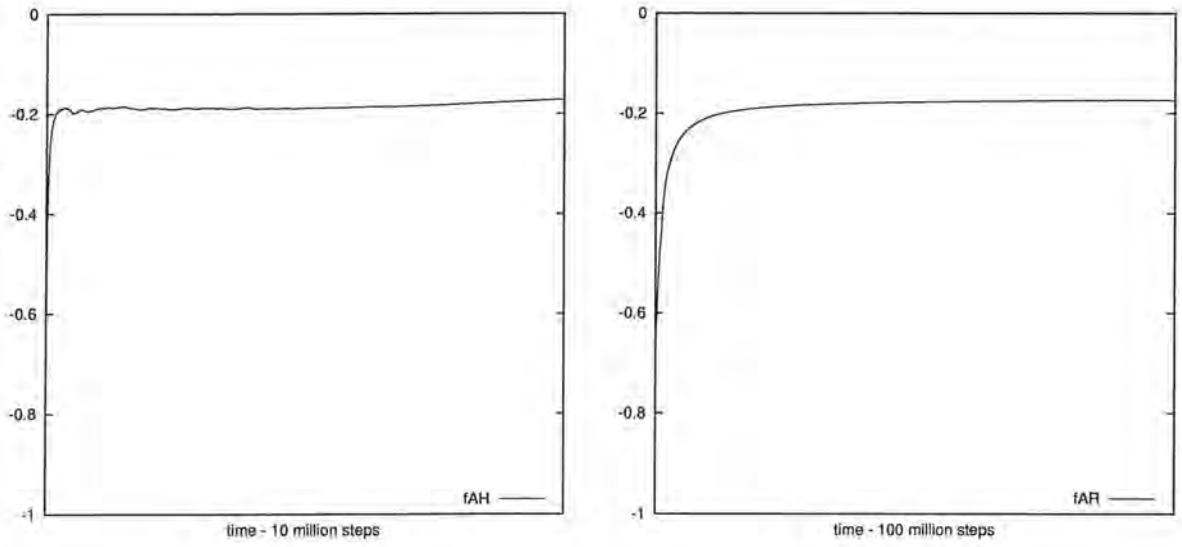


(a) (b)

FIGURE 7.3. AH and AR-Learning in the product delivery environment with 2 trucks, 2 shops, and 5 different different inventory levels.

Due to extremely finicky parameters and the sub-optimal behavior of function-approximated AR-Learning in environments with binary features, (which we did not immediately recognize as unique to lesser quantities of features), we experienced severe difficulties in getting function-approximated AR-Learning to reach an average reward comparable to the one learned by function-approximated AH-Learning. Noting similar difficulties in the adjustment of the parameters alpha and beta in R-Learning in [10], we tried a method similar (but not equivalent) to Mahadevan's table-based beta [10] in our function-approximated AR-Learning. Mahadevan used a table of learning parameters rather than a single variable beta to multiply the different members of the value function. Our 'beta-table' was indexed by feature and value of that feature. Even with this adjustment, function-

approximated AR-Learning did not work as well as function-approximated AH-Learning. However, it scaled much better than the latter method (which is understandable, seeing as it requires less space for keeping models and also fewer calculations) and ran in one-third to one-quarter of the time that AH-Learning required.



(a) (b)

FIGURE 7.4. Function approximated versions AH and AR-Learning for 2 trucks, 5 shops, and 5 possible inventory levels.

By decreasing the value of beta dramatically (to somewhere between 10^{-4} and 10^{-6}), keeping the beta fixed, and increasing the number of iterations which the AR algorithm was run by tenfold, function-approximated AR-Learning was finally able to attain around the same average reward as function-approximated AH-Learning. Figure 7.4 (note part (a) has and x-range of 10 million steps whereas part (b) has an x-range of 100 million steps) shows a comparison between function-approximated AR (notated as fAR) and function-approximated AH (notated as

fAH). The curve that is labeled 'fAR-beta' represents function-approximated AR-Learning, using the beta adjustment mentioned above. Each curve represents an average of 10 runs and is carried out in an environment containing 2 trucks, 5 shops, and 5 different possible levels of both shop inventory and truck loads. This is the same environment attempted in [22], which exhibited the abnormality of AR-Learning that we set out to investigate. The only difference between the environment of the shown plot and the environment of the mentioned study is the probability function representing how often a shop's inventory might decrease.

8. CONCLUSIONS AND FUTURE WORK

Auto-exploratory R-Learning is a relatively simple algorithm. It gets rid of the need for inflicting exploration using an ϵ-greedy step as well as the random steps and sub-optimal returns that come as a result. AR-Learning, as does its non-auto-exploratory counterpart, R-Learning, suffers from a sensitivity of parameters that can be both frustrating and time-consuming. Compared to AH-Learning, AR-Learning also takes more iterations to converge. AR-Learning's improvement in latter iterations is deceptively slow, and therefore seems to be halted earlier than it actually converges. Even though the number of iterations required to reach ultimate convergence is nearly ten times that of AH-Learning, AR-Learning still requires less temporal time to run, due to the smaller number of computations and the lesser number of tables required by this algorithm. These differences are based on the fact that AH-Learning is a model-based method, while AR-Learning is model-free.

We noted a severe difference between table-based and function-approximated AR-Learning as to the difficulty in discovering the parameters that would yield optimal results. Perhaps this difficulty was caused, in part, by the function approximation method. A potential area for future work might include attempting different function approximation methods with AR-Learning.

BIBLIOGRAPHY

[1] L. Baird, "Residual Algorithms: Reinforcement Learning with Function Approximation," *International Conference on Machine Learning*, 1995.

[2] A.G. Barto, S.J. Bradtke, S.P. Singh, "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 73(1), pp.81-138, 1995.

[3] J.A. Boyan, A.W. Moore,"Generalization in Reinforcement Learning: Safely Approximating the Value Function," *Advances in Neural Information Processing Systems*, vol. 7, 1995.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, Second Edition, The MIT Press, 2001.

[5] T.G. Dietterich, Notes and slides from CS 534, Machine Learning, Oregon State University, 2003.

[6] T.G. Dietterich, "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition," *Journal of Artificial Intelligence Research*, vol. 9, pp.227-303, 2000.

[7] T.G. Dietterich, "An Overview of MAXQ Hierarchical Reinforcement Learning," Oregon State University, 2000.

[8] T. Estlin, I. Volpe, D. Mutz, F. Fisher, B. Engelhardt, S. Chien, "Decision-Making in a Robotic Architecture for Autonomy," *JPL online library*, Jet Propulsion Laboratory, 2001.

[9] L.P. Kaelbling, M.L. Littman, A.W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol. 4, pp.237-285, 1996.

[10] S. Mahadevan, "Average Reward Reinforcement Learning: Foundations, Algorithms, and Empirical Results," *Machine Learning*, vol. 22, pp.159-196, 1996.

[11] S. Mahadevan, "To Discount or Not to Discount in Reinforcement Learning: A Case Study Comparing R Learning and Q Learning", *International Conference on Machine Learning*, pp.164-172, 1994.

[12] D. Ok P. Tadepalli, "Auto-exploratory Average Reward Reinforcement Learning," *Proceedings, 14th International Conference on Artificial Intelligence*, pp.881-888, 1996.

[13] R. Parr, S. Russell, "Reinforcement Learning with Hierarchies of Machines," *Advances in Neural Information Processing Systems*, vol. 10, pp.1043-1049, 1997.

[14] S. Proper, P. Tadepalli, H. Tang, R. Logendran, "A Reinforcement Learning Approach for Product Delivery by Multiple Vehicles," Oregon State University.

[15] D.E. Rumelhart, G.E. Hinton, R.J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, p.533-536, 9 October, 1986.

[16] S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Inc., 1995.

[17] A. Schwartz, "A Reinforcement Learning Method for Maximizing Undiscounted Rewards," *Proceedings of the Tenth International Conference on Machine Learning*, 1993.

[18] S. Seri, P. Tadepalli, "Model-based Hierarchical Average-reward Reinforcement Learning," Oregon State University, 2002.

[19] R.S. Sutton, "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding," *Advances in Neural Information Processing Systems*, vol. 8, pp.1038-1044, 1996.

[20] R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 2000.

[21] P. Tadepalli, D. Ok, "Model-based Average Reward Reinforcement Learning," *Artificial Intelligence*, vol. 100, pp.177-224, 1998.

[22] H. Tang, "Average-reward Reinforcement Learning for Product Delivery by Multiple Vehicles," MS Project Report, Oregon State University, 2002.

[23] G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Communications of the ACM*, vol. 38(3), 1995.

[24] S. Thrun, A. Schwartz, "Issues in Using Function Approximation for Reinforcement Learning," Proceedings of the Fourth Connectionist Models Summer School, 1996.

[25] J.N. Tsitsiklis, B. Van Roy, "Feature-Based Methods for Large Scale Dynamic Programming," *Machine Learning*, vol. 22, pp.59-94, 1996.

[26] C.J. Watkins, "Learning from Delayed Rewards," Ph.D. thesis, King's College, 1989.

[27] E.W. Wolfram, "MathWorld – A Wolfram Web Resource", *http://mathworld.wolfram.com*.