Machine Learning, 8, 293–321 (1992)

Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching

LONG-JI LIN ljl@cs.cmu.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213

Abstract. To date, reinforcement learning has mostly been studied solving simple learning tasks. Reinforcement learning methods that have been studied so far typically converge slowly. The purpose of this work is thus two-fold: 1) to investigate the utility of reinforcement learning in solving much more complicated learning tasks than previously studied, and 2) to investigate methods that will speed up reinforcement learning.

This paper compares eight reinforcement learning frameworks: *adaptive heuristic critic* (*AHC*) learning due to Sutton, *Q-learning* due to Watkins, and three extensions to both basic methods for speeding up learning. The three extensions are experience replay, learning action models for planning, and teaching. The frameworks were investigated using connectionism as an approach to generalization. To evaluate the performance of different frameworks, a dynamic environment was used as a testbed. The environment is moderately complex and nondeterministic. This paper describes these frameworks and algorithms in detail and presents empirical evaluation of the frameworks.

Keywords. Reinforcement learning, planning, teaching, connectionist networks

1. Introduction

Reinforcement learning is an interesting learning technique. It requires only a scalar reinforcement signal as performance feedback from the environment. Reinforcement learning often involves two difficult subproblems. The first is known as the *temporal credit assignment problem* (Sutton, 1984). Suppose a learning agent performs a sequence of actions and finally obtains certain outcomes. It must figure out how to assign credit or blame to each individual situation (or situation-action pair) to adjust its decision making and improve its performance. The second subproblem is the *generalization problem* (also known as the *structural credit assignment problem*). When the problem space is too large to explore completely, a learning agent must have the ability to guess about new situations based on experience with similar situations. In the course of learning, both subproblems must be solved.

The most popular and best-understood approach to the temporal credit assignment problem is *temporal difference* (TD) methods (Sutton, 1988). TD methods, which have a solid mathematical foundation, are closely related to dynamic programming (Barto et al., 1991). Two forms of TD-based reinforcement learning have been proposed: the *adaptive heuristic critic* (AHC) learning architecture (Sutton, 1984; Barto et al., 1990) and *Q-learning* (Watkins, 1989). While both methods have been applied to solve some simple learning problems (e.g., (Sutton, 1984; Anderson, 1987; Watkins, 1989; Kaelbling, 1990; Sutton, 1990)), they were found to converge slowly.

Several approaches to the generalization problem have been studied for reinforcement learning. Anderson (1987) and Lin (1991a, 1991b, 1991c) have successfully combined TD

methods with the connectionist error backpropagation algorithm (Rumelhart et al., 1986). Watkins (1989) used a CMAC algorithm, Grefenstette et al. (1990) a genetic algorithm method, Mahadevan and Connell (1991) a statistical clustering method, Chapman and Kaelbling (1991) a method similar to decision-tree algorithms, and Moore (1991) a method which is basically look-up tables but which uses variable state resolution.

The goal of this work is to study connectionist reinforcement learning in solving non-trivial learning problems and to study methods for speeding up reinforcement learning. Although this study took the connectionist approach, the algorithms and results presented here appear to be relevant to other generalization methods as well.

This work studied eight frameworks for connectionist reinforcement learning: connectionist AHC- and Q-learning, and their extensions. A number of studies (Anderson, 1987; Chapman & Kaelbling, 1991; Mahadevan & Connell, 1991) have found that AHC- and Q-learning often converge slowly. For speedup, three extensions to both basic methods were investigated in this work: experience replay, learning action models for planning, and teaching.

The different frameworks were evaluated using a nondeterministic dynamic environment as a testbed. The environment consists of four kinds of objects: the agent, stationary food and obstacles, and moving enemies. The task of the agent is to survive in the environment, which is by no means trivial for a knowledge-poor agent, since the agent has to process a large number of input signals, has several actions to choose from, and has multiple goals.

The rest of this paper is organized as follows. Section 2 presents the learning frameworks and algorithms. Section 3 describes the dynamic environment and the survival task. Sections 4 and 5 present the implementation and performance of the learning agents. Section 6 assesses the different learning frameworks by comparing the agents' performance. Section 7 discusses the limitations. Finally, Section 8 concludes the paper by summarizing the lessons learned from this study.

2. Reinforcement learning frameworks

In the reinforcement learning paradigm, a learning agent continually receives sensory inputs from the environment, selects and performs actions to affect the environment, and after each action, receives from the environment a scalar signal called *reinforcement*. The reinforcement can be positive (reward), negative (punishment), or 0. The objective of learning is to construct an optimal *action selection policy* (or simply policy) that maximizes the agent's performance. A natural measure of performance is the *discounted cumulative reinforcement* (or for short, *utility*) (Barto et al., 1990):

$$V_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1}$$

where V_t is the discounted cumulative reinforcement starting from time t throughout the future, r_t is the reinforcement received after the transition from time t to $t + 1$, and $0 \leq \gamma \leq 1$ is a *discount factor*, which adjusts the importance of long-term consequences of actions.

This section describes eight frameworks for connectionist reinforcement learning. They are summarized as follows:

- AHCON: connectionist AHC-learning
- AHCON-R: AHCON plus experience replay
- AHCON-M: AHCON plus using action models
- AHCON-T: AHCON plus experience replay plus teaching
- QCON: connectionist Q-learning
- QCON-R: Q-CON plus experience replay
- QCON-M: QCON plus using action models
- QCON-T: QCON plus experience replay plus teaching

The basic idea behind these frameworks is to learn an *evaluation function* (see *eval* and *util* below) to predict the discounted cumulative reinforcement to be received in the future. The evaluation function is represented using connectionist networks and learned using a combination of temporal difference (TD) methods (Sutton, 1988) and the error backpropagation algorithm (Rumelhart et al., 1986). In essence, TD methods compute the error (called the *TD error*) between *temporally successive predictions*, and the backpropagation algorithm minimizes the error by modifying the weights of the networks.

Before starting to discuss the learning frameworks, two terms are defined for later use:

- *eval*(x): the expected discounted cumulative reinforcement that will be received starting from world state x, or simply the utility of state x;
- *util*(x, a): the expected discounted cumulative reinforcement that will be received after the execution of action a in response to world state x; or simply the utility of the state-action pair (x, a).

In a deterministic world, *util*(x, a) is equal to the immediate reinforcement r plus the utility of the next state y, discounted by γ:

$$util(x, \mathrm{a}) = r + \gamma \cdot eval(y) \tag{2}$$

Equation 2 can be generalized to a nondeterministic world by taking into consideration the probabilities of multiple outcomes (Watkins, 1989). Note that it is only meaningful to estimate both functions, *eval* and *util*, relative to some policy, since different policies will result in different reinforcements received. Unless explicitly mentioned, both functions are assumed to be relative to the current policy.

2.1. AHC-learning: Framework AHCON

Framework AHCON (Figure 1) is a connectionist implementation of the adaptive heuristic critic learning architecture (Barto et al., 1990). It consists of three components: an *evaluation network*, a *policy network*, and a *stochastic action selector*. In essence, the framework decomposes reinforcement learning into two subtasks. The first subtask is to construct,

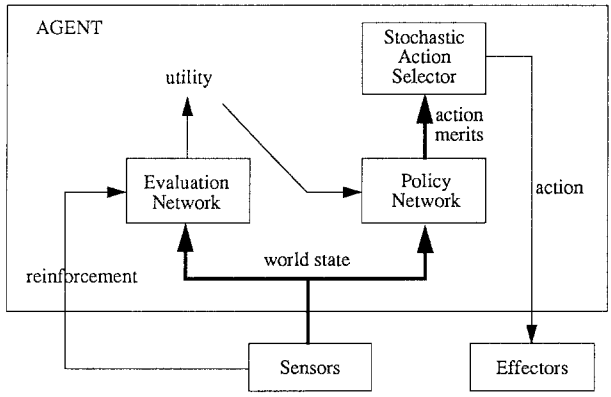

Figure 1. Framework AHCON. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

using TD methods, an evaluation network that accurately models the function $eval(x)$. The policy network takes a world state[1] as inputs and assigns to each action a value (called *action merit*) indicating the relative merit of performing that action in response to that world state. Thus, the second subtask is to adjust the policy network so that it will assign higher merits to actions that result in higher utilities (as measured by the evaluation network).

Ideally, the outputs of the policy network will approach the extremes, for example, 1 for the best action(s) and -1 for the others. Given a policy network and a state, the agent's policy is to choose the action that has the highest merit. In order to learn an optimal policy effectively, the agent has to take actions that appear suboptimal once in a while so that the relative merits of actions can be assessed. A stochastic action selector, which favors actions having high merits, can be used for this purpose.

In the course of learning, both the evaluation and policy networks are adjusted incrementally. Figure 2 summarizes the learning algorithm, in which the simplest TD method, TD(0), is used.[2] To use TD methods to learn an evaluation function, the first step is to write down a recursive definition of the desired function. By Definition 1, the utility of a state x is the immediate payoff r plus the utility of the next state y, discounted by γ.[3] Therefore, the desired function must satisfy Equation 3:

$$eval(x) = r + \gamma \cdot eval(y) \tag{3}$$

During learning, the equation may not hold true. The difference between the two sides of the equation (called TD error) is reduced by adjusting the weights of the evaluation network using the backpropagation algorithm (Step 5).

The policy network is also adjusted (Step 6) according to the same TD error. The idea is as follows: If $e' > e$, meaning that action a is found to be better than previously expected, the policy is modified to increase the merit of the action. On the other hand, if $e' < e$, the policy is modified to decrease the merit. The policy network is not updated with respect to actions other than a, since from a single experience we know nothing about the merits of the other actions.

1. $x \leftarrow$ current state; $e \leftarrow eval(x)$;
2. $a \leftarrow select(policy(x), T)$;
3. Perform action a; $(y, r) \leftarrow$ new state and reinforcement;
4. $e' \leftarrow r + \gamma \cdot eval(y)$;
5. Adjust evaluation network by backpropagating TD error $(e' - e)$ through it with input x;
6. Adjust policy network by backpropagating error Δ through it with input x, where $\Delta_i = \begin{cases} e' - e & \text{if } i = a \\ 0 & \text{otherwise} \end{cases}$
7. go to 1.

Figure 2. The algorithm for Framework AHCON.

During learning, the stochastic action selector (i.e., the *select* function in the algorithm) chooses actions randomly according to a probability distribution determined by action merits. In this work, the probability of choosing action a_i is computed as follows:

$$Prob(a_i) = e^{m_i/T} / \sum_k e^{m_k/T} \tag{4}$$

where m_i is the merit of action a_i, and the *temperature* T adjusts the randomness of action selection.

2.2. *Q-learning: Framework QCON*

Framework QCON (Figure 3) is a connectionist implementation of *Q-learning* (Watkins, 1989). QCON learns a network (called the *utility network*) that accurately models the function *util*(x, a). Given a utility network and a state x, the agent's policy is to choose the

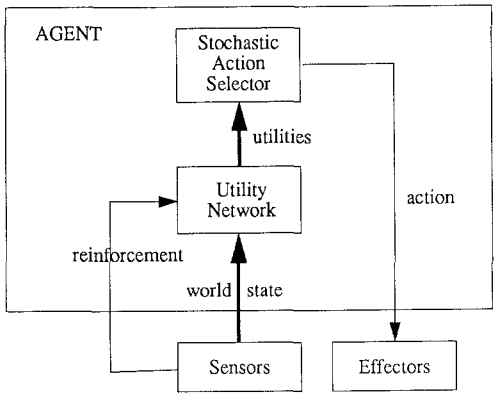

Figure 3. Framework QCON. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

1. $x \leftarrow$ current state; for each action i, $U_i \leftarrow util(x, i)$;
2. $a \leftarrow select(U, T)$;
3. Perform action a; $(y, r) \leftarrow$ new state and reinforcement;
4. $u' \leftarrow r + \gamma \cdot Max\{util(y, k) \mid k \in actions\}$;
5. Adjust utility network by backpropagating error ΔU through it with input x, where $\Delta U_i = \begin{cases} u' - U_i & \text{if } i = a \\ 0 & \text{otherwise} \end{cases}$
6. go to 1.

Figure 4. The algorithm for Framework QCON.

action a for which $util(x, a)$ is maximal. Therefore, the utility network is not only an evaluation function but also used as a policy.

The learning algorithm is depicted in Figure 4 and briefly described below. Generally speaking, the utility of an action a in response to a state x is equal to the immediate payoff r plus the best utility that can be obtained from the next state y, discounted by γ. Therefore, the desired *util* function must satisfy Equation 5:

$$util(x, a) = r + \gamma \cdot Max\{util(y, k) \mid k \in actions\} \tag{5}$$

During learning, the difference between the two sides of the equation is minimized using the backpropagation algorithm (Step 5). Note that the network is not modified with respect to actions other than a, since from a single experience we know nothing about the utilities of the other actions.

The utility network described above has multiple outputs (one for each action). It could be replaced by multiple networks (one for each action); each with a single output. The former implementation might be less desirable, because whenever the single utility network is modified with respect to an action, no matter whether it is desired or not, the network is also modified with respect to the other actions as a result of shared hidden units between actions. A similar argument can apply to the policy network of Framework AHCON. (This study used multiple networks with single outputs.)

Again, a stochastic action selector is needed to explore the consequences of different actions. The same action selector used by Framework AHCON can be employed for this purpose.

2.3. *Experience replay: Frameworks AHCON-R and QCON-R*

Reinforcement learning is a trial-and-error method, and thus may result in undesirable damages to the learning agent in a hostile environment. The faster the agent learns, the less damage the agent is likely to suffer. Generally speaking, temporal difference learning is a slow process for temporal credit assignment, especially when credits must be propagated through a long action sequence. The rest of this section describes three techniques to speed up the credit propagation process and/or to shorten the trial and error process.

The basic AHC- and Q-learning algorithms described above are inefficient in that experiences obtained by trial-and-error are utilized to adjust the networks only once and then thrown away. This is wasteful, since some experiences may be rare and some (such as those involving damages) costly to obtain. Experiences should be reused in an effective way.

In this paper, an *experience* is a quadruple, (x, a, y, r), meaning that the execution of an action a in a state x results in a new state y and reinforcement r. A *lesson* is a temporal sequence of experiences starting from an initial state to a final state, where the goal may or may not be achieved.

The most straightforward way of reusing experiences is what I call *experience replay*. By experience replay, the learning agent simply remembers its past experiences and repeatedly presents the experiences to its learning algorithm as if the agent experienced again and again what it had experienced before. The result of doing this is that the process of credit/blame propagation is sped up and therefore the networks usually converge more quickly. However, it is important to note that a condition for experience replay to be useful is that the laws that govern the environment of the learning agent should not change over time (or at least not change rapidly), simply because if the laws have changed, past experiences may become irrelevant or even harmful.

Experience replay can be more effective in propagating credit/blame if a sequence of experiences is replayed in temporally **backward** order. It can be even more effective if TD(λ) methods are used with the *recency factor*, λ, greater than zero. With $\lambda > 0$, the TD error for adjusting network weights is determined by the discrepancy between not only two but multiple consecutive predictions. Lin (1991c) presents a detailed algorithm and study of using backward replay and nonzero λ in a mobile robot domain. The results of that paper are encouraging. In this paper, however, only backward replay and $\lambda = 0$ are used for simplicity reason.

The algorithm for AHCON-R is simply the one for AHCON plus repeated presentation of past experiences to the algorithm, except that *experiences involving non-policy actions (according to the current policy) are not presented*. By presenting an experience to the algorithm, I mean to bind the variables, (x, a, y, r), used in the algorithm with the experience. The reason for the exception is the following: AHCON estimates $eval(x)$ (relative to the current policy) by sampling the current policy (i.e., executing some policy actions). Replaying past experiences, which is equivalent to sampling policies in the past, will disturb the sampling of the current policy, if the past policies are different from the current one. Consider an agent whose current policy chooses a very good action a in state x. If the agent changes to choose a very bad action b in the same state, the utility of state x, $eval(x)$, will drop dramatically. Similarly, the utility of x will be underestimated if the bad action b are replayed a few times. For experience replay to be useful, the agent should only replay the experiences involving actions that still follow the current policy.

Just like AHCON-R but for a different reason, QCON-R should also replay only actions which follow the current policy. Consider the previous example. Even if the agent changes from choosing good action a to choosing bad action b in state x, the values of both $util(x, a)$ and $util(x, b)$ will not change. As a matter of fact, if the *util* function is implemented as look-up tables, Watkins (1989) has shown that Q-learning is guaranteed to learn an optimal policy as long as all state-action pairs are tried an infinite number of times (along with a few weak conditions). In other words, it is not harmful for the tabular version of Q-learning

to replay bad actions. But this does not hold true for connectionist Q-learning, because whenever the backpropagation algorithm modifies a utility network with respect to one input state, it also affects the network with respect to many or all of the possible input states. If a utility network is trained on bad experiences many times consecutively, the network might come to underestimate the real utilities of some state-action pairs.

In short, AHCON-R and QCON-R should replay only policy actions. However, since an agent may use a stochastic policy as suggested previously, all actions have more or less possibility of being chosen by any policy. It is difficult to say whether an action is a current policy action or not. In this work, an action is considered as non-policy action, if its probability (see Equation 4) of being chosen (according to the *current* policy) is lower than P_l.[4] For AHCON-R, $P_l = 0.2$ was found to give roughly the best performance for the learning task to be described later in this paper. For QCON-R P_l, = 0.1, 0.01 and 0.001 were tried and found to give similar performance. ($P_l = 0$ indeed gave bad performance.) In other words, AHCON-R is quite sensitive to replaying non-policy actions, while QCON-R is much less sensitive. The difference in sensitivity can be explained by the fact that replaying non-policy actions, in the case of using look-up tables, is bad for AHC-learning but is fine for Q-learning.

For Frameworks AHCON-R and QCON-R to be efficient in terms of time and memory space, only "recent" experiences need to be stored and replayed. As a matter of fact, it is not only unnecessary but also harmful to replay an experience as many times as possible, because the networks might be over-trained and become too specific to that experience, which usually harms generalization. Consider an agent living in a nondeterministic world. Whenever the agent is in state x and performs action a, 80% of the time it will receive a great penalty, while 20% of the time it will receive no penalty. If the agent just experienced the situation with no penalty and is trained on this experience many times, the agent will come to believe that it is harmless to perform a in x, which is certainly wrong. Section 4.7 describes a partial solution to prevent over-training.

2.4. Using action models: Frameworks AHCON-M and QCON-M

Another way of reusing past experiences, which was investigated in the Dyna architecture (Sutton, 1990), is to use experiences to build an *action model*[5] and use it for planning and learning. An action model is a function from a state and an action, (x, a), to the next state and the immediate reinforcement, (y, r). For stochastic worlds, each action may have multiple outcomes. To be accurate, the action model may also need to model the probability distribution of outcomes, which is an issue this paper does not address.

An action model is intended to mimic the behaviors of the environment. Using an accurate action model (if it is available), the agent can experience the consequences of actions without participating in the real world. As a result, the agent will learn faster (if projecting is faster than acting) and more importantly, fewer mistakes will be committed in the real world. Frameworks AHCON-M and QCON-M to be discussed below are based on this idea.

2.4.1. *Framework AHCON-M*

Framework AHCON-M is very similar to Framework AHCON, but differs in that AHCON-M also learns an action model and uses it to do what Sutton called *relaxation planning* (Sutton, 1990). Relaxation planning, which is closely related to dynamic programming, is an incremental planning process that consists of a series of shallow (usually one-step look-ahead) searches and ultimately produces the same results as a conventional deep search. In Sutton's Dyna architecture, his learning algorithm (similar to the one for AHCON) is applied to real-world situations faced by the agent and also hypothetical situations randomly generated. In the former case the next state is obtained by executing an action, while in the latter case the next state is obtained by applying an action model. His approach has two inefficiencies. First of all, since hypothetical situations are randomly generated, the agent may spend too much effort planning what to do about hypothetical situations that will never happen in the real world at all. Second, it is not clear how his approach decides when relaxation planning is no longer necessary and to stop doing it.

The relaxation planning algorithm (see Figure 5) proposed here addresses both inefficiencies by projecting all actions from states actually visited, not from states chosen at random. Since all actions to a state are examined at the same time, the relative merits of actions can be more directly and effectively assessed than the kind of *policy iteration* (Howard, 1960) used in AHCON and the Dyna architecture. Compared with the Dyna architecture, the disadvantage of this algorithm, however, is that the number of hypothetical experiences that can be generated is limited by the number of states visited.

In a deterministic world, the utility of a state x is equal to the immediate payoff r obtained from executing the best action a plus the discounted utility of the new state y:

$$eval(x) = Max\{r + \gamma \cdot eval(y) \mid a \in actions\} \quad \text{(Note: } y \text{ is a function of } a\text{)} \quad (6)$$

Thus, if a correct action model is available, the utility of a state can be accurately estimated by looking ahead one step. During learning, the evaluation network is adjusted to minimize the difference between the two sides of Equation 6 (Step 6).

The policy network is updated in the following manner: First we compute the average utility of state x, μ, under the assumption that the current policy and stochastic action selector will be used throughout the future (Step 5). *Prob* is the probability distribution function for action selection; it is Equation 4 in this work. Next, the policy network is modified to increase/decrease the merits of actions which are above/below the average (Step 7).

The algorithm for Framework AHCOM-M is similar to that for AHCON, except that relaxation planning may take place either before Step 2 or after Step 3 (in Figure 2)—in the latter case the agent will be more reactive, while in the former case the agent may make better action choices because it can benefit directly from the one-step look-ahead. To be efficient, relaxation planning is performed selectively (Steps 2 & 3 in Figure 5). The idea is this: For situations where the policy is very decisive about the best action, relaxation planning is not needed. If the policy cannot be very sure about which is the best action, relaxation planning is performed. In this way, at the beginning of learning, all actions are equally good and relaxation planning is performed frequently. As learning proceeds, relaxation planning is performed less and less often. (In this study, promising actions are those whose probability of being chosen is greater than 2%.)

1. $x \leftarrow$ current state; $e \leftarrow eval(x)$;
2. Select promising actions S according to $policy(x)$;
3. If there is only one action in S, go to 8;
4. For $a \in S$ do
   4.a. Simulate action a; $(y, r) \leftarrow$ predicted new state and reinforcement;
   4.b. $E_a \leftarrow r + \gamma \cdot eval(y)$;
5. $\mu \leftarrow \sum_{a \in S} Prob(a) \cdot E_a$; $max \leftarrow Max\{E_a \mid a \in S\}$;
6. Adjust evaluation network by backpropagating error $(max - e)$ through it with input x;
7. Adjust policy network by backpropagating error Δ through it with input x, where $\Delta_a = \begin{cases} E_a - \mu & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$
8. exit.

Figure 5. Relaxation planning algorithm for AHCON-M. *Prob* is the probability function for stochastic action selection.

2.4.2 *Framework QCON-M*

Framework QCON-M is Framework QCON plus using action models. The algorithm is similar to that in Figure 4. The main difference is that before Step 2 or after Step 3, the agent can perform relaxation planning by looking one step ahead (see Figure 6). For the same reason that it is harmful for QCON-R to replay non-policy actions (Section 2.3), experiencing bad actions with a model is also harmful.[6] Therefore, relaxation planning must be performed selectively. For situations where the best action is obvious, no look-ahead is needed. If there are several promising actions, then these actions are tried with the model. QCON-M is similar to Sutton's Dyna-Q architecture except that only the currently visited state is used to start hypothetical experiences. (Again, in this study, promising actions are those whose probability of being chosen is greater than 2%.)

1. $x \leftarrow$ current state; for each action i, $U_i \leftarrow util(x, i)$;
2. Select promising actions S according to U;
3. If there is only one action in S, go to 6;
4. For $a \in S$ do
   4.a. Simulate action a; $(y, r) \leftarrow$ predicted new state and reinforcement;
   4.b. $U'_a \leftarrow r + \gamma \cdot Max\{util(y, k) \mid k \in actions\}$;
5. Adjust utility network by backpropagating error ΔU through it with input x, where $\Delta U_a = \begin{cases} U'_a - U_a & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$
6. exit.

Figure 6. Relaxation planning algorithm for QCON-M.

2.5. Teaching: Frameworks AHCON-T and QCON-T

Reinforcement learning is a trial-and-error process. The success of learning relies on the agent's luck in achieving the goal by chance in the first place. If the probability of achieving the goal by chance is arbitrarily small, the time needed to learn will be arbitrarily long (Whitehead, 1991b). This learning barrier will prevent agents from shortening learning time dramatically on their own. One way (probably the best way) to overcome the barrier is to learn expertise directly from external experts.

Teaching plays a critical role in human learning. Very often teaching can shorten our learning time, and even turn intractable learning tasks into tractable. If learning can be viewed as a search problem (Mitchell, 1982), teaching, in some sense, can be viewed as external guidance for this search. Teaching is useful for two reasons: First, teaching can direct the learner to first explore the promising part of the search space which contains the goal states. This is important when the search space is large and thorough search is infeasible. Second, teaching can help the learner avoid being stuck in local maxima. A real example of this is presented in Section 6.4.

While learning from experts is an efficient way to learn, experts may not be available, or may be available only part of the time. So, after having overcome the most formidable learning barrier with the help of experts, the learners must be able to work out the rest on their own. As will be shown below, teaching can be easily and gracefully integrated into both AHC- and Q-style of reinforcement learning. The frameworks to be discussed below are able to learn from teachers as well as through reinforcement learning.

Frameworks AHCON-T and QCON-T, which are AHCON-R and QCON-R plus teaching, use exactly the experience replay algorithms for AHCON-R and QCON-R, respectively. Teaching is conducted in the following manner: First, the teacher shows the learning agent how an instance of the target task can be achieved from some initial state. The sequence of the shown actions as well as the state transitions and received reinforcements are recorded as a *taught lesson*. Several taught lessons can be collected and repeatedly replayed the same way *experienced* (i.e., self-generated) *lessons* are replayed. Like experienced lessons, taught lessons should be replayed selectively; in other words, only policy actions are replayed. But if the taught actions are known to be optimal, all of them can be replayed all the time. In this paper, the term *lesson* means both taught and experienced lessons.

It is unnecessary that the teacher demonstrate only optimal solutions in order for the agent to learn an optimal policy. In fact, the agent can learn from both positive and negative examples. This property makes this approach to teaching different from the *supervised learning* approaches (e.g., (Mozer, 1986; Pomerleau, 1989)). In the supervised learning paradigm, a learning agent tries to mimic a teacher by building a mapping from situations to the demonstrated actions and generalizing the mapping. The consequent drawbacks are that (1) the teacher is required to create many many training instances to cover most of the situations to be encountered by the agent, (2) when a new situation is encountered and the agent does not have a good strategy for it, a teacher must be available to give the solution, and (3) the teacher must be an expert, or else the agent cannot learn the optimal strategy. The third drawback is often neglected by researchers, but it is important when humans want to build robots using supervised learning techniques. Since humans and robots have different sensors and hence see different things, an optimal action from a human's

point of view may not be optimal for robots if we take into account the fact that robots may not be able to sense all the information that humans use to make decisions. Human teachers must teach in terms of what robots can sense.

Obviously, Frameworks AHCON-T and QCON-T do not have the first and second drawbacks. They also do not have the third drawback, because they do not learn by rote. Instead, they determine the real utilities of the shown actions—If a shown action results in a state which the agent **now** believes to be good/bad, then the agent increments/decrements the likelihood of performing that action given the same situation in the future. On the other hand, typically learners are going to benefit from experts more than from naive teachers.

3. A dynamic environment

To date, reinforcement learning has been studied mostly for solving simple learning tasks, such as pole-balancing, route-finding, etc. (Some researchers, for instance, Mahadevan and Connell (1991) and Chapman and Kaelbling (1991), have begun to study more complicated problems, however.) It is therefore unclear whether reinforcement learning can scale up to deal with more realistic problems. One of the goals of this work is thus to study the utility of various frameworks in solving nontrivial learning problems. To this end, I devised a simulated dynamic environment to be used as a testbed for evaluating the performance of various frameworks.

The dynamic environment is a 25 × 25 cell world. A sample environment is shown in Figure 7.[7] There are four kinds of objects in the environment: the agent ("I"), food ("$"), enemies ("E"), and obstacles ("O"). The perimeter of the world is considered to be occupied by obstacles. The bottom of the figure is an energy indicator ("H"). At the start, the agent and four enemies are placed in their initial positions as shown in Figure

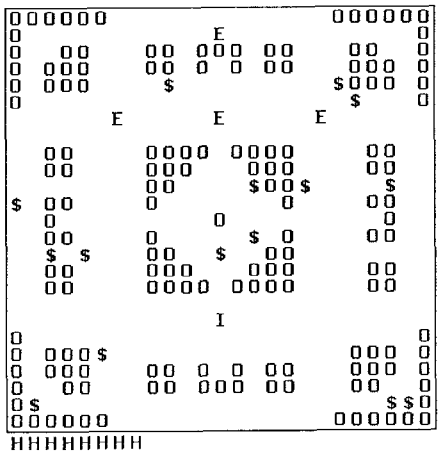

Figure 7. A dynamic environment involving an agent (I), enemies (E), food ($), and obstacles (O). The H's indicate the agent's energy level.

7, and fifteen pieces of food are randomly placed on unoccupied cells. On each move, the agent has four actions to choose from; it can walk to one of the four adjacent cells. If the agent attempts to walk into obstacles, it will remain at the same position.

After the agent moves, each of the enemies is allowed to stay or move to an adjacent cell that is not occupied by obstacles. To allow the agent to escape from chasing enemies, the enemy speed is limited to 80% of the full speed of the agent. The enemies move randomly, but tend to move towards the agent; the tendency becomes stronger as the agent gets closer. Appendix A gives the algorithm for choosing enemy actions.

The agent has two goals: to get as much food as possible and to avoid being caught by enemies. Note that the two goals conflict in some situations, and the agent must learn to arbitrate between them. A play ends when the agent gets all of the food or dies. The agent dies when it either collides with an enemy or runs out of energy. At the start of a new play, the agent is given 40 units of energy. Each piece of food provides the agent 15 units of additional energy, and each move costs the agent 1 unit. These parameter values were empirically chosen so that survival in the environment would not be too easy or too difficult.

To make the learning task more interesting and realistic, the agent is allowed to see only a local area surrounding it. From the agent's point of view, the world is nondeterministic, not only because the enemies behave randomly, but also because the world is only partially observable (thus, what will be seen after a move is not completely predictable). Although a human player can avoid the enemies and get all of the food most of the time, survival in this environment is not trivial. To survive, the agent must learn to 1) approach food, 2) escape from enemies, 3) avoid obstacles, 4) identify and stay away from certain dangerous places, (e.g., corridors), where it can be easily seized, and 5) seek food when food is out of sight.

4. The learning agents

This section presents the implementation of four AHC-agents (i.e., Agents AHCON, AHCON-R, AHCON-M, and AHCON-T) and four Q-agents (i.e., Agents QCON, QCON-R, QCON-M, and QCON-T), which learn to survive in the environment. The agents are named after the learning frameworks on which they are based. In order to compare their performance, the agents use exactly the same reinforcement signals and sensory inputs as described below.

All of the connectionist networks are trained using a symmetrical version of the error backpropagation algorithm; the squashing function is a variation of the sigmoid function: $f(x) = 1/(1 + e^{-x}) - 0.5$. (Note: Once the agents die, they get re-incarnated to try again, and in the meanwhile their learned networks are preserved.)

4.1. The reinforcement signals

After each move, the learning agent receives from the environment one of the following reinforcement signals:

- −1.0 if the agent dies,
- 0.4 if the agent gets food,
- 0.0 otherwise.

Negative reinforcement is considered bad, and positive is good. The food reward is smaller than the penalty for being dead, since it is more important to stay alive. The food reward, 0.4, was chosen empirically; I tried a few different values for Agent AHCON, and 0.4 gave roughly the best performance. Theoretically speaking, the agent should learn to get food even without food reward, because the agent is required to get food to stay alive. But the sooner good action decisions are rewarded, the sooner the agent should learn.

4.2. Input representation

As described before, AHC-agents have evaluation and policy networks, while Q-agents have utility networks. This subsection describes the input representation for the networks, while another subsection describes the output representation.

The evaluation, policy and utility networks are all structurally similar; each of them is a three-layer, feed-forward network, consisting of 145 input units and 1 output unit. (The number of hidden units is a parameter to be tuned for performance.) The networks are fully connected except that there are no connections between the input and output units. The input units of the networks can be divided into five groups: enemy map, food map, obstacle map, energy level, and some history information. Each of the maps shows a certain kind of object in a local region surrounding the agent and can be thought of as being obtained by an array of sensors fixed on the agent. The sensor array moves as the agent moves. If the local action representation (see Section 4.3) is used, the array also rotates as the agent rotates. Figure 8 shows the configuration of the food, enemy and obstacle sensor arrays.

Each food sensor may be activated by several nearby food objects and each food object may activate several nearby food sensors. The food sensor array is composed of three different

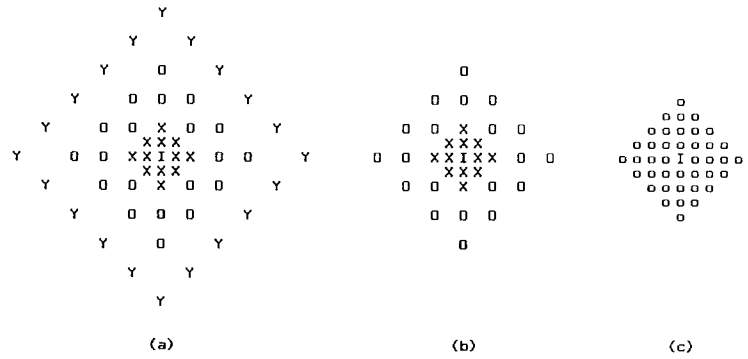

Figure 8. The (a) food, (b) enemy, and (c) obstacle sensor arrays.

types of sensors, types "X", "O", and "Y". Different food sensor types have different resolution and different receptive fields—"X", "O", and "Y" sensors can be activated by food at any of the five, nine, and thirteen nearby cells, respectively. This technique of encoding spatial positions of objects is known as *coarse coding* (Hinton et al., 1986). Through the use of multiple resolution and coarse coding techniques, the food positions are effectively coded without loss of critical information.

The enemy sensor array has a similar layout of the food sensor array, except that it consists of only "X" and "O" types of sensors. The obstacle sensors are organized differently; there is only one type of obstacle sensor, and each obstacle sensor is only activated by an obstacle at the corresponding cell. The coarse coding technique is not used to encode obstacle positions, because it only works effectively when the features to be encoded are sparse (Hinton et al., 1986), and this is not the case for obstacles.

The agent's energy level is, again, coarse-coded using sixteen input units. Each of the sixteen units represents a specific energy level and is activated when the agent's energy level is close to that specific level. Finally, four units are used to encode the agent's previous action choice and one unit to indicate whether or not the previous action resulted in a collision with an obstacle. These five units convey a kind of history information, which allows the agent to learn heuristics such as "moving back to previous positions is generally bad," "when no interesting objects (e.g., food) are around, keep moving in the same direction until you see something interesting," etc.

4.3. Action representation

As mentioned before, the agent has four actions. This work experimented with two different representations of agent actions:

- **Global representation:** The four actions are to move to the north, south, west and east, regardless of the agent's current orientation in the environment.
- **Local representation:** The four actions are to move to the front, back, right and left relative to the agent's current orientation.

If the local representation is used, the agent begins with a random orientation for each new play, and its orientation is changed to the move direction after each step. For instance, if the agent takes four consecutive "right" moves and no collision occurs, it will end up being at the same cell with the same orientation as before.

4.4. Output representation

The single output of the evaluation network represents the estimated utility of the given input state. Corresponding to the action representations, there are two different output representations for policy and utility networks.

Global representation: With this action representation, the agent uses only one policy network, whose single output represents the merit of moving to the "north." The merits of moving in other directions can be computed using the same network by rotating the state inputs (including food map, enemy map, obstacle map, and the 4 bits encoding the agent's previous action) by 90, 180, and 270 degrees. Similarly, the agent uses only one utility network, whose single output represents the utility of moving to the "north" in response to the input state. Again, the utilities of moving in other directions can be computed by rotating the state inputs appropriately. By taking advantage of the action symmetry, the learning task is simplified, because whatever is learned for a situation is automatically carried over to situations which are more or less symmetric to it.

Local representation: This representation does not take advantage of action symmetry. Each AHC-agent uses four policy networks and each Q-agent four utility networks. Each network has a single output.

The outputs of all the networks are between −1 and +1. By Definition 1, the *eval* and *util* functions may be greater than 1 when the agent is close to many pieces of food. In such (rare) cases, they are truncated to 1 before being used to compute TD errors. Also, the output units of the evaluation and utility networks use mainly the linear part of the sigmoid function. An alternative is to use a truly linear activation function for the output units.

4.5. The action model

Agents AHCON-M and QCON-M learn an action model. The action model is intended to model the input-output behavior of the dynamic environment. More specifically, given a world state and an action to be performed, the model is to predict what reinforcement signal will be received, where the food, enemies and obstacles will appear, and what the agent's energy level will be. In this nondeterministic environment, each action can have many possible outcomes. There are two alternatives to modeling the environment: the action model can generate either a list of outcomes associated with probabilities of happening or only the most likely outcome. The second alternative is adopted, because of its simplicity.

Since the food and obstacles do not move, their positions were found to be quite easy to predict using connectionist networks. So, to shorten simulation time without significantly simplifying the model-learning task, Agents AHCON-M and QCON-M were only required to learn *reinforcement networks* for predicting the immediate reinforcement signal and *enemy networks* for predicting the enemy positions. The reinforcement and enemy networks are all two-layer networks (i.e., without hidden layers). The reinforcement networks use as inputs all of the 145 input bits mentioned before, while the enemy networks use only the enemy and obstacle maps. The single output of each enemy network is trained to predict whether a particular enemy sensor will be turned on or off after the agent moves.

The reinforcement and enemy networks are learned on-line just like the other networks. Learning these networks is a kind of supervised learning, and the encountered experiences (i.e., x, a, y, r) can be saved in a queue (of limited length) for repeated presentation to the networks. Because the enemies are nondeterministic and the agent does not know the exact enemy positions due to coarse coding, the prediction of enemy positions was found to be often incorrect even after the networks were well-trained. Since it is also interesting

to see what performance the agent will have if a perfect model can be learned quickly, AHCON-M and QCON-M are also provided with a "perfect" model, which is just the environment simulator. Section 6.3 presents a performance comparison between using a perfect model and using a learned, potentially incorrect one.

4.6. Active exploration

An important problem often faced by autonomous learning agents is the tradeoff between acting to gain information and acting to gain rewards. Kaelbling (1990) proposed an *interval estimate* (IE) method for control of this tradeoff. However, it is unclear whether the method can apply to connectionist reinforcement learning. Thrun and Möller (1992) proposed to have the agent learn a *competence map*, which estimates the errors of what the agent learned about the world. The map should display small errors for well-explored world states and large errors for rarely-explored states. Active exploration is achieved by driving the agent into regions of large errors.

As mentioned before, this work uses a stochastic action selector as a crude strategy to allow active exploration. It was found that the stochastic selector alone did not give a good compromise between gaining information and gaining rewards. A complementary strategy is also used: the agent dead-reckons its trajectory and increases the temperature T whenever it finds itself stuck in a small area without getting food. A similar strategy is also used for some robot learning tasks (Lin, 1991c). For both domains, this strategy was found quite effective in improving learning speed and quality.

4.7 Prevention of over-training

As mentioned in Section 2.3, undesired over-training may occur if the same experiences are replayed too many times, no matter whether the experiences come from a teacher or from trial-and-error. To reduce the possibility of over-training and also to save computation, the following heuristic strategies are employed:

- After each complete play, the agent replays n lessons chosen randomly from the most recent 100 experienced lessons, with recent lessons exponentially more likely to be chosen. n is a decreasing number between 12 and 4. Appendix B gives the algorithm for choosing experiences stochastically.
- After each complete play, the agent (if it is AHCON-T or QCON-T) also stochastically chooses taught lessons for replay. Each of the taught lessons is chosen with a decreasing probability between 0.5 and 0.1.

Recall that only policy actions are replayed. Policy actions are those whose probability of being chosen is greater than $P_l = 0.2$ for AHCON-type agents or $P_l = 0.01$ for Q-type agents. (Since the taught lessons I gave to the agents were nearly optimal, P_l was in fact set to 0 during replaying taught lessons.)

5. Experimental results

This section presents the performance of various learning agents. In the first study, the learning agents used the global action representation and took advantage of action symmetry. In the second study, the agents used the local action representation and did not exploit action symmetry.

5.1. Experimental design

Each study consisted of 7 experiments. (Each experiment took a Sparc Station two days to complete.) For each experiment, 300 training environments and 50 test environments were randomly generated. The agents were allowed to learn only from playing the 300 training environments. Each time an agent played 20 training environments, it was tested on the 50 test environments with learning turned off and the temperature T set to zero. The average number of food pieces obtained by the agent in the test environments was then plotted versus the number of training environments that the agent had played so far. The learning curves presented below show the mean performance over all the experiments.

It is instructive to mention the difference between the objective of the learning algorithms and that I set up for the agents. The former is to maximize the discounted cumulative reinforcement, while the latter is to have as much food as possible. Achieving the former objective does not necessarily end up achieving the latter. However, given the reinforcement function defined in Section 4.1, it is unlikely that both objectives are quite different.

For both studies, two lessons were generated for Agents AHCON-T and QCON-T to use. To generate the lessons, I pretended to be the agent trying to survive in a manually-chosen environment, which involved a few instructive situations. I got all the food in each of the lessons.

Each learning agent has several parameters that can be tuned for performance:

- γ: the discount factor (fixed to be 0.9);
- T: the temperature for the stochastic action selector;
- H_e, H_p and H_u: the number of hidden units of the evaluation, policy and utility networks;
- η_e, η_p and η_u: the learning rate of the backpropagation algorithm for the evaluation, policy and utility networks;
- the momentum factor of the backpropagation algorithm (fixed to be 0.9 for all networks);
- the range of the random initial weights of networks (fixed to be 0.1).

5.2. Study 1: Using global action representation

In this study, the agents used the global action representation and exploited action symmetry. Table 1 shows the parameter settings used to generate the mean learning curves shown in Figure 9.[8] Those parameter values were empirically chosen to give roughly the best performance for Agents AHCON and QCON. Little search was done for the other agents. AHC-agents used a temperature much lower than that used by Q-agents, because action

Table 1. Parameter values used for Study 1.

Agent	H	η	T
AHCON	$H_e = 30$ $H_p = 30$	$\eta_e = 0.2$ $\eta_p = 0.4$	$1/T = 2 \rightarrow 10$
Other AHC-agents	$H_e = 30$ $H_p = 30$	$\eta_e = 0.1$ $\eta_p = 0.2$	$1/T = 2 \rightarrow 10$
QCON	$H_u = 30$	$\eta_u = 0.3$	$1/T = 20 \rightarrow 60$
Other Q-agents	$H_u = 30$	$\eta_u = 0.15$	$1/T = 20 \rightarrow 60$

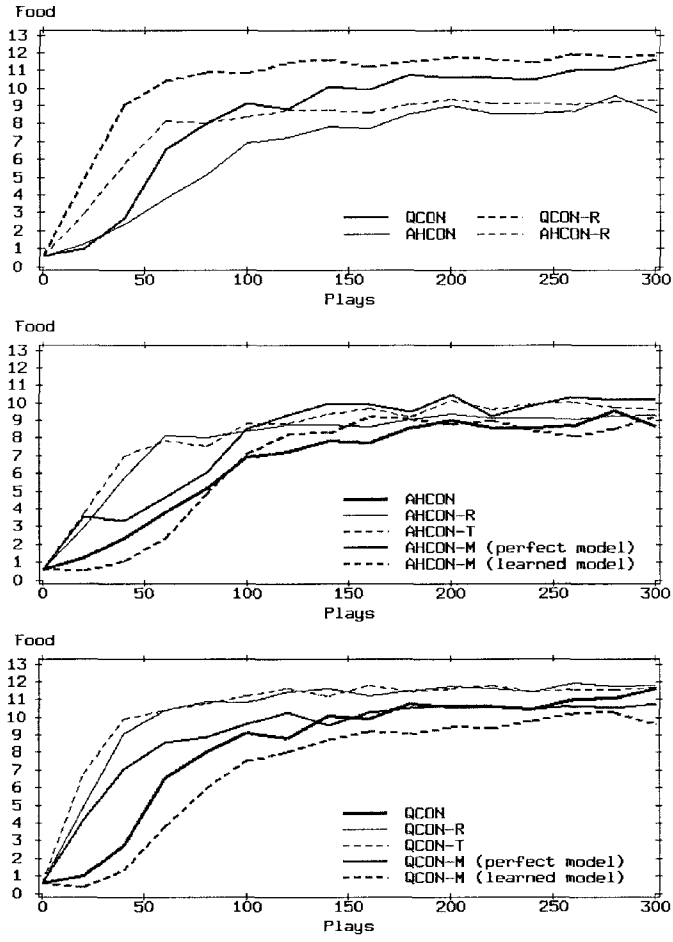

Figure 9. Learning curves of the agents using the global action representation.

merits are supposed to approach 1 for the best action(s) and −1 for the others, while utilities of state-action pairs are usually small numbers between 1 and −1. Cooling temperatures were used, although fixed low temperatures seemed to work as well. The learning agents with learning-speedup techniques used smaller learning rates than those used by the basic learning agents, because the former trained their networks more often.

5.3. *Study 2: Using local action representation*

In this study, the agents used the local action representation and did not exploit action symmetry. The agents used the same parameter settings as in Study 1, except that all the learning rates were doubled. Figure 10 shows the learning curves of the agents.

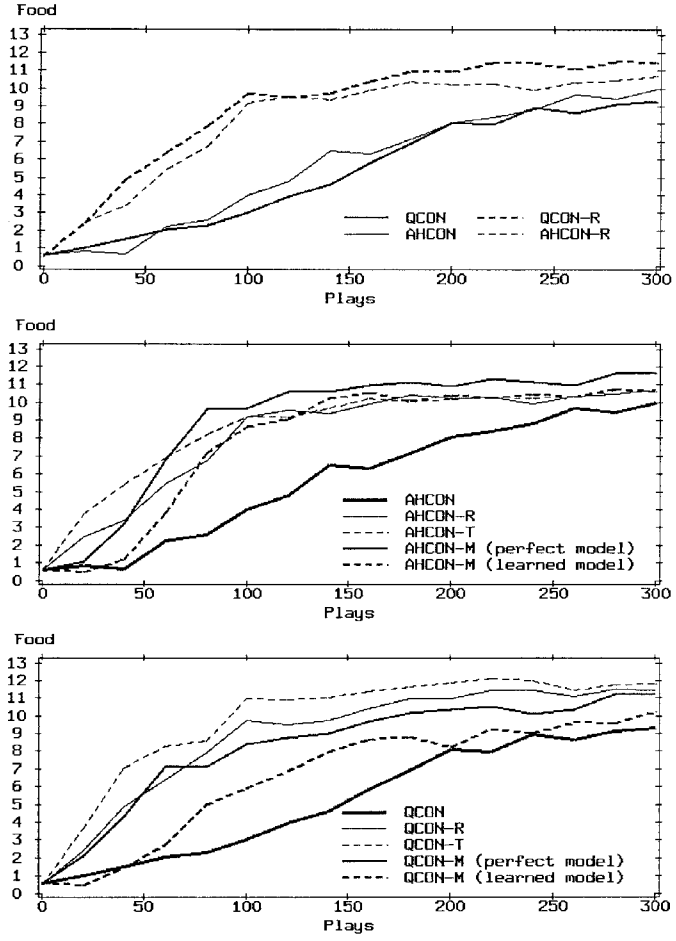

Figure 10. Learning curves of the agents using the local action representation.

5.4. More observation

Among all the learning agents, QCON-T was roughly the best one. To see the absolute performance of the best agent after 300 trials, QCON-T was tested on 8000 randomly generated environments. The following table shows how often the agent got all the food, got killed, or ran out of energy.

got all food	got killed	ran out of energy
39.9%	31.9%	28.2%

In average the agent got 12.1 pieces of food in each play. The table below shows how often the agent got no food, 1 piece of food, more pieces, or all of the 15 pieces.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
%	0.1	0.3	0.8	1.8	2.2	2.9	4.0	4.1	3.8	3.7	3.4	4.1	5.4	8.2	15.2	39.9

As mentioned previously, the learning curves in Figures 9 and 10 show the mean performance over 7 experiments. The standard deviations of these curves, point by point, are roughly 2 food pieces at the beginning of learning and 1 food piece after the asymptotic performance is reached. Generally speaking, the agents who had better performance had smaller performance deviations between experiments. But an agent's performance difference between experiments did not imply much about the repeatability of experimental results, because each experiment used a different set of training and test environments. What really matters is the relative performance of agents, which was found consistent most of the time through each entire experiment.

6. Discussion

This section compares the learning agents by analyzing the experimental results in Section 5.

6.1. AHC-agents vs. Q-agents

In the first study, the asymptotic performance of Q-agents was significantly better than that of AHC-agents. A number of previous studies (Lin, 1991a; Sutton, 1990) also found the superiority of Q-learning over AHC-learning. But Study 2 did not confirm this superiority; in fact both types of learning were found to be similarly effective in terms of both asymptotic performance and learning speeds. It is still an open question whether Q-learning works better than AHC-learning in general.

Note also that the two different action representations did not make a significant difference for Q-agents. But for AHC-agents, the local action representation apparently worked better than the global action representation, as far as asymptotic performance is concerned. This result reveals that different action representations, as well as different input representations, may make a big difference in the performance of a learning system.

6.2. Effects of experience replay

Compared with AHCON (or QCON), the learning speed of AHCON-R (or QCON-R) was significantly improved. The difference in their asymptotic performance, however, was not found to be significantly different after 300 trials in Study 1, and seemed to get less significant in Study 2 if learning continued. This concludes that experience replay is indeed an effective way to speed up the credit assignment process. Experience replay is also very easy to implement. The cost of using it is mainly the extra memory needed for storing experiences. To prevent over-training and to be efficient, a learning agent also needs a good strategy for determining what experiences to remember and how often to replay them.

6.3. Effects of using action models

The advantage of using action models for relaxation planning was inconclusive. On the one hand, compared with AHCON (or QCON), the learning speed of AHCON-M (or QCON-M) was significantly improved when the agent did not need to learn an action model on its own (i.e., a perfect model was provided). On the other hand, when the agent needed to learn a model itself, whether using a model was going to help depended on the relative difficulty of learning the model and learning the survival task directly. In the second study, learning the model was found to be faster than learning the task. Therefore once a sufficiently good model had been learned, Agents AHCON-M and QCON-M began to benefit from using it. On the other hand, in the first study AHCON and QCON already could learn the task quite rapidly. Therefore a potentially incorrect model turned out to be only misleading, at least in the beginning. As a matter of fact, QCON-M with a learned model performed worse than QCON in Study 1, which can be explained by the fact that the agent was not able to learn a perfect model for the nondeterministic world, even after hundreds of trials.

As speculated in Section 2.4, relaxation planning should be less and less needed as learning proceeds. This speculation was confirmed in both studies. Figure 11 shows the average number of hypothetical actions that were taken on each step. (The curves show the mean

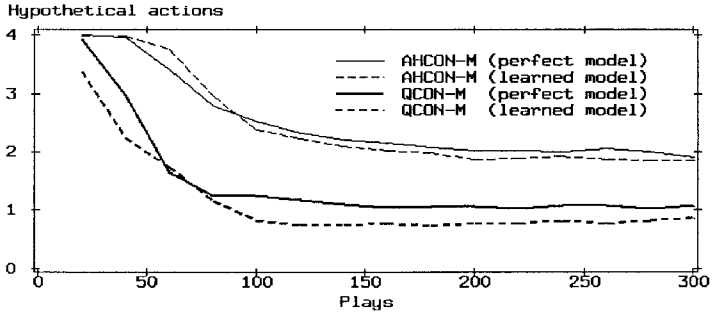

Figure 11. Average number of hypothetical actions taken on each step (from Study 2).

results from the 7 experiments of Study 2.) The maximum is 4, since there are 4 actions to choose from on any step. As we can see from the figure, the amount of planning dropped very quickly as learning approached the asymptote. But it did not drop close to zero at the end, because in this task there are many situations where several actions are in fact almost equally good and therefore they all end up being tried in simulation. For tasks where only a small portion of actions are relevant most of the time, the saving should be more significant.

6.4. Effects of teaching

Recall that AHCON-T (or QCON-T) is AHCON-R (or QCON-R) plus replaying taught lessons. In the first study, the performance differences between AHCON-T and AHCON-R and between QCON-T and QCON-R were negligible over the entire experiment. Since the task in Study 1 was not very difficult and experience replay alone had done a good job, there was not much improvement we could expect from using teaching. But in the second study, there were noticeable differences between the learning speeds of AHCON-R and AHCON-T and between those of QCON-R and QCON-T. The differences between their asymptotic performance, however, were insignificant.

This result reveals that the advantage of teaching should become more significant as the learning task gets more difficult. Indeed, Lin (1991c) reported similar results from a learning robot. One of the robot's tasks is to dock on a battery charger, using sonar sensors for detecting obstacles and light intensity sensors for locating the charger, which has a light on top of it. To dock, the robot must position itself in front of the charger within small errors and then drive to collide with the charger in order to get a tight connection. Without a teacher, the robot failed to learn the task, because at the early stage of learning, the robot quickly learned to avoid obstacles, which in turn prevented it from colliding with the charger (the learning was stuck in a local maximum). After being shown a lesson (i.e., how to dock from one sample position), the robot became able to learn the task. After being shown four lessons, the robot learned the task rapidly.

6.5. Experience replay vs. using action models

Comparing the learning curves of Agents AHCON-M and AHCON-R and those of QCON-M and QCON-R, there was clear, consistent superiority of experience replay over using action models, when the agents had to learn a model themselves. (But there was no clear, consistent superiority of one over the other, when the agents were provided with a perfect action model.) This result can be simply explained by the fact that it took some time for Agents AHCON-M and QCON-M to learn a sufficiently good model before the agents could start taking advantage of it. Before a good model was learned, the model could be misleading and impede learning. (As a matter of fact, AHCON-M and QCON-M did not perform relaxation planning for the first 10 trials, avoiding using a completely senseless model.)

Why couldn't relaxation planning with a perfect action model outperform experience replay? First of all, the relaxation planning algorithms used here may not be the most effective

way of using action models. For example, the relaxation planning algorithms used here performed just one-step look-ahead. Also, the number of hypothetical experiences that AHCON-M and QCON-M could generate was limited by the number of states actually visited. (On the other hand, if the agents are allowed to generate hypothetical experiences from any randomly chosen states, they may spend too much time planning what to do about hypothetical situations that will never happen at all.)

Secondly, *experience replay is also a kind of relaxation planning*! Experience replay uses an "action model," but does not need to learn one; an action model is simply obtained by sampling past experiences. In essence, the collection of past experiences is a model. It represents not only explicitly the environment's input-output patterns but also implicitly the probability distributions of multiple outcomes of actions.

Since experience replay is so effective and also easy to implement, is it of no use to learn an action model? The answer is unclear. If a model is learned *merely* for doing relaxation planning, perhaps it is *not* worthwhile to learn a model, since experience replay does the same thing as relaxation planning. One may argue that: 1) a model that generalizes can provide the learning algorithms with induced or interpolated experiences, which are not seen before, and 2) the extra experiences will result in better learning of the evaluation functions. But if the evaluation functions also generalize (as it should be the case for non-toy tasks), it is unclear whether these extra experiences can actually do any extra good.

On the other hand, having an action model can be useful. In this work I only investigated how to use action models for learning the evaluation functions. There are other ways of using a model to improve performance that I did not study here. For example, we can use an evaluation function, an action model, and a look-ahead planning technique to help find the best actions (Whitehead & Ballard, 1989; Thrun et al., 1991). In a complex domain where an optimal policy may be hardly obtainable, by looking ahead a few steps (much as computer chess does), the non-optimality of a policy can be compensated, if an accurate action model is available. How to use action models effectively is an interesting issue and needs further study.

6.6. Why not perfect performance?

Why did all the agents fail to reach the perfect performance (i.e., get all the food)? There are at least two reasons: 1) The local information used by the agents may be insufficient to determine the optimal actions, and 2) the perfect policy may be too complex to be represented by the connectionist networks used here. As a matter of fact, I also played against the simulator myself. Being allowed to see only objects in the local region as the learning agents do, I got all the food pieces most (but not all) of the time. I found that I often employed two techniques to play it so successfully: look-ahead planning (although very crude) and remembering food positions so that I could come back to get the food after I lost the sight of food because of chasing enemies. The learning agents did not use either of the two techniques.

7. Limitations

While it seems that all of these learning frameworks are promising, they have several common limitations.

Representation dependent. As in any other learning system, a good input representation is critical to successful learning. Because of the use of coarse coding and concentric multiple resolution maps, the number of input units needed for the task was dramatically reduced. Without using these techniques, the agents could not have been so successful. Choosing a good action representation is also crucial sometimes. For example, the local action representation gave AHC-agents better performance than the global action representation.

Discrete time and discrete actions. So far TD methods have been used almost exclusively in domains with discrete time and actions. While it seems possible to extend the idea of temporal difference to handle continuous time and actions, it is not clear how easily this extension can be done. But note that these frameworks can work for continuous states, since connectionist networks take real as well as binary numbers as inputs.

Unwise use of sensing. The learning agent is required to sense the entire environment at each step to determine what world state it is in. In the real world, there are too many things to look at. Most of them may be irrelevant. In practice, an agent often cannot afford to sense everything all the time. It is an important issue to decide how to use sensors efficiently while still knowing what the current state is (Whitehead & Ballard, 1991a; Tan, 1991).

History insensitive. Unlike humans who usually make decisions based on information which is currently sensed and information which is in the past and cannot be sensed, the learning agent is only reactive to what is perceived at the current moment, but insensitive to what was perceived in the past. One possibility of becoming history-sensitive is to use *time-delay networks* (Lang, 1989) or *recurrent networks* (Williams & Zipser, 1988).

Perceptual aliasing. The *perceptual aliasing problem* occurs when the agent's internal state representation is insufficient to discriminate different external world states. As a consequence of perceptual aliasing, the learning agent cannot learn a correct evaluation function and therefore cannot act optimally. Consider a packing task which involves three steps: open a box, put an object in the box, and close the box. An agent driven only by its current visual percepts cannot accomplish this task, because facing a closed box, the agent does not know whether an object is already in the box or not. There are two ways to resolve this problem. The first solution is to actively choose perceptual actions, such as measuring the weight of the box, to resolve the ambiguity. This kind of solutions have been investigated by Whitehead and Ballard (1991a) for a block-stacking problem and by Tan (1991) for a route-finding problem. The second solution is to use history information, such as whether the box was ever opened or not, to help determine the current state of the world. This solution seems more general and more powerful than the first one. To deal with complex real world problems, perhaps both kinds of solutions are needed.

No hierarchical control. In theory, TD methods are able to accurately propagate credit through a long sequence of actions. However, this can be true only when the evaluation function can be modeled to any arbitrary accuracy, for example, using look-up tables. In practice, more compact function approximators that allow generalization, such as connectionist networks and decision trees, should be used. By using such approximators, TD methods can hardly propagate credit accurately through an action sequence longer than,

say, 20. Furthermore, the longer the sequence is, the more time is needed for credit propagation. A potential solution to this problem is to use *hierarchical control*. By hierarchical control, a top-level task is decomposed into subtasks, each subtask is learned separately using reinforcement learning, and finally a top-level policy is learned to control the invocation of the subtasks (Lin, 1991c; Mahadevan & Connell, 1991).

8. Conclusion

This paper studied connectionist AHC- and Q-learning for a moderately complex task—survival in a dynamic environment. The results are encouraging and suggest that reinforcement learning is a promising approach to building autonomous learning systems. Some previous studies (Lin, 1991a; Sutton, 1990) found the superiority of Q-learning over AHC-learning. This work confirmed this finding in one case and found comparable performance of both methods in the other. More studies will be required before it is possible to formulate the conditions for which AHC- or Q-learning will be more effective than the other.

Reinforcement learning algorithms often converge slowly. This paper investigated three extensions for speedup: experience replay, learning action models for relaxation planning, and teaching. Relaxation planning, which is a kind of incremental dynamic programming (Watkins, 1989), caches the results of repeated shallow searches in an evaluation function, using an action model for looking ahead. Learning an action model for planning is a well-known idea. But whether using a model is going to speed up learning depends on the relative difficulty of learning the model and learning to solve the task directly. In this study, learning a good model for the nondeterministic, dynamic world turned out to be difficult, and this has limited the utility of using a model in some cases.

By sampling past experiences, experience replay performs a process much like relaxation planning, but does not need to learn an action model. The key point is to replay only policy actions. This study found that experience replay was quite effective in speeding up the credit assignment process. As a matter of fact, experience replay was found to work better than relaxation planning with a learned model; after all, learning a model takes time. So, perhaps it is *not* worthwhile to learn a model, if a model is learned *merely* for doing relaxation planning. But an action model is still a good thing to have, because a model together with an evaluation function can be utilized, for instance, to perform conventional look-ahead search for optimal actions when an optimal policy is not available.

This paper also described an approach to integrating teaching with reinforcement learning. To learn to solve a problem by reinforcement learning, the learning agent must achieve the goal (by trial-and-error) at least once. Teaching can be an effective technique to shorten the trial-and-error process by simply providing some success examples to the learner. Teaching can also help the learner avoid being stuck in local maxima (for instance, Section 6.4). Although teaching was not found to be critical to learning the survival task in this work, I expect teaching to be more and more important as tasks get more complicated and rewards get less likely to obtain by luck (Lin, 1991c).

Acknowledgments

I would like to express my gratitude to Tom Mitchell and Rich Sutton for many fruitful discussions on this work. Thanks also to Jeffrey Schlimmer, Nils Nilsson, the reviewers, and Sebastian Thrun for their valuable comments. This work was supported partly by NASA under Contract NAGW-1175 and partly by Fujitsu Laboratories Ltd.

Notes

1. Normally the agent's internal description of the world is obtained from a vector of sensory readings, which may or may not be pre-processed. Here it is assumed that the agent's input adequately represents the external world and is sufficient to determine the optimal actions. Reinforcement learning often works more or less, even when this assumption is relaxed.
2. See (Lin, 1991c); Watkins, 1989; Dayan, 1992) for using TD(λ), $\lambda > 0$.
3. For simplicity, I will assume the world is deterministic throughout the discussion of the learning algorithms, although it is straightforward to extend the discussion to handle nondeterministic worlds (Barto et al., 1990; Watkins, 1989). The reinforcement learning algorithms presented in the paper can work for deterministic and nondeterministic worlds.
4. Another way to reduce the negative effects of AHCON-R replaying non-policy actions is to treat an action being a policy action as a matter of probability. In other words, the computed TD error, which is used to adjust the networks, is weighted by the probability of choosing the replayed action. This method, however, did not work as effectively as the thresholding method (i.e., using P_l).
5. Instead of using the term *world model* as Sutton did, I decided to use the term *action model* to refer to a model for predicting the effects of actions. I will use the term *world model* to mean an agent's internal representation of the world. For example, knowledge about the next-state function is referred to as action model, while a cognitive map about a familiar environment is referred to as world model.
6. It seems that AHCON-M does not have this problem, since at each planning step, only the most promising action is used to determine the amount of change to *eval*(x).
7. It may appear that the learning task is too simplified by making the obstacles symmetric. That is not all true, because there are also other objects in the world—enemies and food pieces are positioned randomly and with no symmetrical pattern. The number of different input patterns that the agent is likely to come across is estimated to be greater than 2^{50}.
8. My previous studies (Lin, 1991a; Lin, 1991b) reported similar results. There are in fact a few differences between the experimental designs of this study and the previous studies. For example, the action model was different, some of the parameter values were changed, and some implementation details of the learning algorithms were also different. See (Lin, 1991a; Lin 1991b) for more details about the differences.

References

Anderson, C.W. (1987). Strategy learning with multilayer connectionist representations. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 103–114).

Barto, A.G., Sutton, R.S., & Watkins, C.J.C.H. (1990). Learning and sequential decision making. In: M. Gabriel & J.W. Moore (Eds.), *Learning and computational neuroscience*. MIT Press.

Barto, A.G., Bradtke, S.J., & Singh, S.P. (1991). *Real-time learning and control using asynchronous dynamic programming*. (Technical Report 91-57). University of Massachusetts, Computer Science Department.

Chapman, D. & Kaelbling, L.P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. *Proceedings of IJCAI-91*.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning, 8*, 341–362.

Grefenstette, J.J., Ramsey, C.L., & Schultz, A.C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning, 5*, 355–382.

Hinton, G.E., McClelland, J.L., & Rumelhart, D.E. (1986). Distributed representations. *Parallel distributed processing: Explorations in the microstructure of cognition, Vol. 1*, Bradford Books/MIT Press.

Howard, R.A. (1960). *Dynamic programming and Markov processes*. Wiley, New York.

Kaelbling, L.P. (1990). *Learning in embedded systems*. Ph.D. Thesis, Department of Computer Science, Stanford University.

Lang, K.J. (1989). *A time-delay neural network architecture for speech recognition*. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University.

Lin, Long-Ji. (1991a). Self-improving reactive agents: Case studies of reinforcement learning frameworks. *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats* (pp. 297–305). Also Technical Report CMU-CS-90-109, Carnegie Mellon University.

Lin, Long-Ji. (1991b). Self-improvement based on reinforcement learning, planning and teaching. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 323–327).

Lin, Long-Ji. (1991c). Programming robots using reinforcement learning and teaching. *Proceedings of AAAI-91* (pp. 781–786).

Mahadevan, S. & Connell, J. (1991). Scaling reinforcement learning to robotics by exploiting the subsumption architecture. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 328–332).

Mitchell, T.M. (1982). Generalization as search. *Artificial Intelligence*, 18, 203–226.

Moore, A.W. (1991). Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 333–337).

Mozer, M.C. (1986). *RAMBOT: A connectionist expert system that learns by example*. (Institute for Cognitive Science Report 8610). University of California at San Diego.

Pomerleau, D.A. (1989). *ALVINN: An autonomous land vehicle in a neural network* (Technical Report CMU-CS-89-107). Carnegie Mellon University.

Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning internal representations by error propagation. *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1.* Bradford Books/MIT Press.

Sutton, R.S. (1984). *Temporal credit assignment in reinforcement learning*. Ph.D. Thesis, Dept. of Computer and Information Science, University of Massachusetts.

Sutton, R.S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9–44.

Sutton, R.S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Workshop on Machine Learning* (pp. 216–224).

Tan, Ming. (1991). Learning a cost-sensitive internal representation for reinforcement learning. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 358–362).

Thrun, S.B., Möller, K., & Linden, A. (1991). Planning with an adaptive world model. In D.S. Touretzky (Ed.), *Advances in neural information processing systems 3*, Morgan Kaufmann.

Thrun, S.B. & Möller, K. (1992). Active exploration in dynamic environments. To appear in D.S. Touretzky (Ed.), *Advances in neural information processing systems 4*, Morgan Kaufmann.

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. Ph.D. Thesis, King's College, Cambridge.

Williams, R.J. & Zipser, D. (1988). *A learning algorithm for continually running fully recurrent neural networks* (Institute for Cognitive Science Report 8805). University of California at San Diego.

Whitehead, S.D. & Ballard, D.H. (1989). A role for anticipation in reactive systems that learn. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 354–357).

Whitehead, S.D. & Ballard, D.H. (1991a). Learning to perceive and act by trial and error. *Machine Learning*, 7, 45–83.

Whitehead, S.D. (1991b). Complexity and cooperation in Q-learning. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 363–367).

Appendix A: The algorithm for choosing enemy actions

Each enemy in the dynamic environment behaves randomly. On each step, 20% of the time the enemy will not move, and 80% of the time it will choose one of the four actions, A_0 (east), A_1 (south), A_2 (west), and A_3 (north), according to the following probability distribution:

$$prob(A_i) = P_i/(P_0 + P_1 + P_2 + P_3)$$

where

$$P_i = \begin{cases} 0 & \text{if } A_i \text{ will result in collison with obstacles} \\ exp(0.33 \cdot W(angle) \cdot T(dist)) & \text{otherwise} \end{cases}$$

$angle$ = angle between the direction of action A_i and the direction from the enemy to the agent

$dist$ = distance between the enemy and the agent

$$W(angle) = (180 - |angle|)/180$$

$$T(dist) = \begin{cases} 15 - dist & \text{if } dist \leq 4 \\ 9 - dist/2 & \text{if } dist \leq 15 \\ 1 & \text{otherwise} \end{cases}$$

Appendix B: The algorithm for choosing lessons for replay

The agents which replay experiences keep only the most recent 100 lessons in memory. Lessons are randomly chosen for replay; recent lessons are exponentially more likely to be chosen. The algorithm for choosing a lesson from memory is shown below.

Input: a sequence of lessons, $L_0, L_1, \ldots, L_{n-1}$, where L_{n-1} is the latest.

Output: an integer k, $0 \leq k < n$

Algorithm:

1. $w \leftarrow Min(3, 1 + 0.02 \cdot n)$
2. $r \leftarrow$ a random number between 0 and 1
3. $k \leftarrow n \cdot log(1 + r \cdot (e^w - 1))/w$