Self-improving Reactive Agents: Case Studies of Reinforcement Learning Frameworks

Long-Ji Lin

13 August 1990

CMU-CS-90-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

To appear in *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, September 24-28, 1990, Paris, France

Abstract

The purpose of this work is to investigate and evaluate different reinforcement learning frameworks using connectionist networks. I study four frameworks, which are adopted from the ideas developed by Rich Sutton and his colleagues. The four frameworks are based on two learning procedures: the Temporal Difference methods for solving the credit assignment problem, and the backpropagation algorithm for developing appropriate internal representations. Two of them also involve learning a world model and using it to speed learning. To evaluate their performance, I design a dynamic environment and implement different learning agents, using the different frameworks, to survive in it. The environment is nontrivial and nondeterministic. Surprisingly, all of the agents can learn to survive fairly well in a reasonable time frame. This paper describes the learning agents and their performance, and summarizes the learning algorithms and the lessons I learned from this study.

This research was supported by NASA under Contract NAGW-1175.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NASA.

Keywords: machine learning, reinforcement learning, connectionist network, temporal difference, world model

Table of Contents

List of Figures

1. Introduction

Reinforcement learning is an interesting learning problem. It requires only a scalar reinforcement signal as a performance feedback from the environment. Reinforcement learning often involves two difficult subproblems. The first is called the *credit assignment problem.* Suppose the 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 arises from the need to develop the appropriate internal representations required to achieve the target learning tasks. In the course of learning, both subproblems must be solved.

Several reinforcement learning frameworks or algorithms have been proposed in the literature (e.g., [Sutton 84, Williams 87, Barto, Sutton & Watkins 89, Watkins 89, Kaelbling 89, Sutton 90]). However, most have only been studied solving simple learning problems (e.g., [Anderson 89]). In addition, no serious comparison of different frameworks has been done. This work is thus intended to be a first step towards the investigation and evaluation of different reinforcement learning frameworks in solving nontrivial learning tasks. In particular, I am interested in reinforcement learning using connectionist networks.

In the paper I study four reinforcement learning frameworks, which are adopted from the ideas developed in [Barto, Sutton & Watkins 89, Watkins 89, Sutton 90]. All of these frameworks are based on two learning procedures: the Temporal Difference (TD) methods [Sutton 88] for solving the credit assignment problem and the error backpropagation algorithm [Rumelhart, et al. 86a] for developing appropriate internal representations.

Generally speaking, reinforcement learning based solely on the TD methods is a slow process. In domains where reinforcements are sparse, the learning rate is slow, and if the cost of mistakes (e.g., physical damage) is also high, the agent would make more mistakes than allowed. A solution to these problems is to learn a world model, and practice with the model. This idea is embodied in two of the frameworks studied here.

My approach to evaluating different learning frameworks is to design a dynamic environment, implement learning agents to survive in it using different frameworks, and evaluate the performance of the agents. Four kinds of objects are involved in this environment: the agent, fixed food and obstacles, and moving enemies. Although survival in this environment is easy for humans, it is by no means trivial for knowledge-poor agents.

The remaining of this paper is organized as follows. Section 2 discusses the four learning frameworks. Section 3 describes the rules of the environment. Sections 4 and 5 present the implementation and performance of the learning agents. Section 6 assesses the merits of the agents. Finally, Section 7 concludes the paper by summarizing the lessons I learned from this study.

2. Reinforcement Learning Frameworks

Learning to survive in an unknown environment can be characterized as a kind of *reinforcement learning*. In reinforcement learning, the 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 objective of learning is to construct an optimal action selection *policy* that maximizes the agent's performance. A natural measure of performance is the

discounted cumulative reinforcements [Barto, Sutton & Watkins 89]:

$$V_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} \quad (1)$$

where V_t is the discounted cumulative reinforcements starting from time t throughout the future, r_{t+1} is the reinforcement received after the transition from time t to t+1, and the *discount rate* γ, $0 \le \gamma < 1$, adjusts the importance of long-term consequences of actions. For short, I use the term *utility* to refer to the discounted cumulative reinforcements. In the paper, both terms are used interchangeably.

In this section, four learning frameworks and the corresponding learning algorithms are described. Basically, the main idea behind the frameworks is to learn an *evaluation function* to predict the discounted cumulative reinforcements to be received. The evaluation function is represented using connectionist networks and is learned using a combination of the Temporal Difference (TD) methods [Sutton 88] and the error backpropagation algorithm [Rumelhart, et al. 86a]. In essence, the TD methods compute the error (called *TD error*) between *temporally successive predictions*, while the backpropagation algorithm tries to minimize the error by changing the weights of the networks.

Before starting to discuss the learning frameworks, here I define two terms for later use:

- *eval(x)*: the discounted cumulative reinforcements to be received starting from world state x, or simply the utility of state x;
- *util(x,a)*: the discounted cumulative reinforcements to be received on the execution of action a in response to world state x; or simply the utility of the pair of state x and action 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:

$$util(x,a) = r + \gamma \cdot eval(y) \quad (2)$$

Equation (2) can be generalized to a nondeterministic world by taking into consideration the probabilities of multiple outcomes.

2.1. Framework A

Framework A (Figure 2-1) was previously studied in [Sutton 84, Barto, Sutton & Watkins 89, Anderson 89]. 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, using the TD methods, an evaluation network that accurately estimates the utility of each world state. The second subtask is to adjust the policy to choose actions leading to high-utility states as measured by the evaluation network. Specifically, the policy network, which takes a world state as inputs, assigns to each possible action a value indicating the relative merit of performing that action in response to that world state. The policy network is adjusted so that actions resulting in higher utilities will be assigned with higher values. So, if such an optimal policy network is available, the best action can be obtained by applying the network to the given state and choosing the action with the highest merit value. To learn an optimal policy effectively, active exploration of different actions in response to the same situation is needed so that the relative merits of actions can be assessed. The stochastic action selector is used for this purpose.

In the course of learning, both the evaluation and the policy networks are adjusted incrementally. Figure 2-2 summarizes the learning algorithm, in which the simplest TD method is used. To use the TD methods to learn an evaluation function, the first step is to write down a recursive definition of the desired function. By definition, the utility of a state x is the immediate payoff r plus the utility of the next state y,

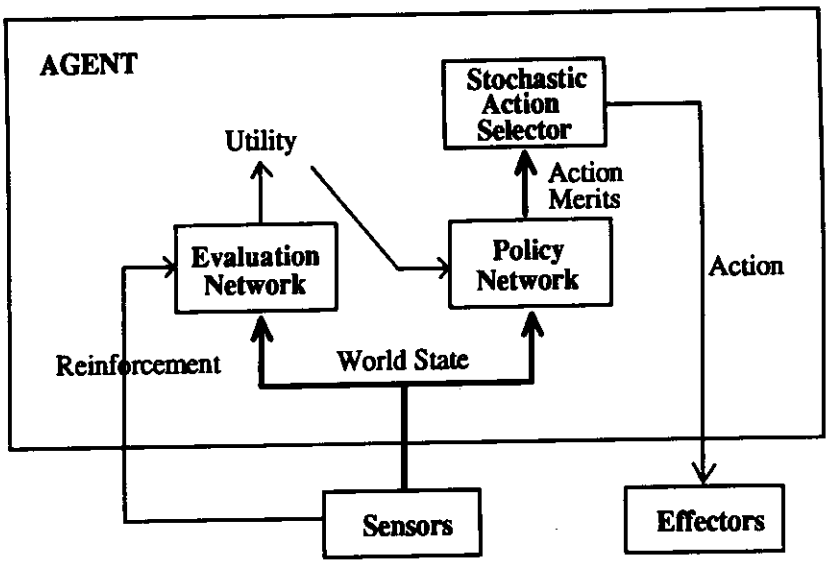

Figure 2-1: Framework A. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

---

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. $e' \leftarrow r + \gamma * eval(y)$;
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. goto 1.

Figure 2-2: The algorithm for Framework A

---

discounted by γ.[1] So, if the prediction is accurate, Equation (3) should hold true:

$$eval(x) = r + \gamma \cdot eval(y) \qquad (3)$$

The TD error is the difference between the two sides of the equation, and the evaluation network is adjusted to reduce the error using the backpropagation algorithm.

The policy network is also adjusted (Step 6) according to the same TD error. The main 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] See [Barto, Sutton & Watkins 89] for a detailed discussion.

In the course of learning, the stochastic action selector (i.e., the *select* function in the algorithm) chooses actions randomly according to a probability distribution determined by the outputs of the policy network-- actions favored by the policy network are more likely to be chosen. In this work, the probability of choosing action a_i is computed according to the following function:

$$Prob(a_i)=\frac{e^{x_i/T}}{\sum_k e^{x_k/T}} \quad (4)$$

where x_i represents the input to the action selector corresponding to action a_i, and the *temperature* T adjusts the randomness of action selection.

2.2. Framework B

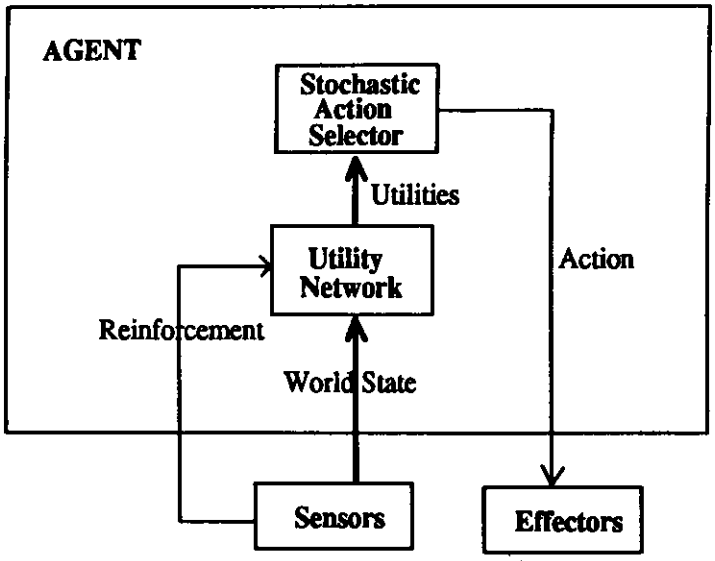

Figure 2-3: Framework B. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

---

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. $u' \leftarrow r + \gamma * Max_{k \in actions}\, util(y,k)$;
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. goto 1.

Figure 2-4: The algorithm for Framework B

Framework B, which is also known as *Q-learning*, was developed by [Watkins 89]. Instead of learning two separate networks as Framework A does, Framework B only learns a single network called the *utility* network. The utility network is like the evaluation network of Framework A in that both are trained to

predict the cumulative reinforcements. However, they differs in that for each action, the utility network maps a world state to the utility of each state-action pair, rather than the utility of the state (recall Equation (2) for the difference). So, if such an optimal utility network is available, the best action can be determined by applying the network to the given state and finding the action that results in the largest value (i.e., utility). Therefore, the utility network is also like the policy network of Framework A in that both are used to choose actions.

The connectionist learning algorithm is depicted in Figure 2-4 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. So, if the prediction is correct, Equation (5) should hold true:

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

The TD error for action a is the difference between the two sides of the equation, and the utility network is modified to reduce this error for action a. However, 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) 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, it is also modified with respect to the other actions as a result of shared hidden units between actions.[2]

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

2.3. Framework AM

Frameworks A and B typically learn by participating in the environment and incrementally modifying their behaviors according to the reinforcement signals. If the cost of making mistakes (e.g., physical damage) is high, learning agents based on the frameworks are subject to suffering. Furthermore, both frameworks mainly rely on the TD methods to propagate back credit or blame (i.e., reinforcements) to appropriate states and/or actions within action sequences. Generally speaking, this propagation process is slow particularly when the reinforcement signals are sparse. A solution to these problems is to use a world model. By using a world model, the agent can learn without participating in the real world, thus learning faster and committing less mistakes in the real world. Frameworks AM and BM to be discussed below are based on this idea.

Framework AM is very similar to Framework A; both consist of an evaluation network, a policy network, and an action selector. They only differ in that Framework AM also learns a world model and uses it to speed learning. This framework was studied in [Sutton 90]. In his work, a world model was merely used as a substitute of the real world, and the algorithm for the framework was exactly the same as the one in Figure 2-2 except that actions were sometimes taken with the real world and sometimes taken with a world model. Here I argue that a world model can be utilized in a more effective way, which is depicted in Figure 2-5 and discussed below.

---

[2] A similar argument can apply to the policy network of Framework A.

1. $x \leftarrow$ current state; $e \leftarrow eval(x)$;
2. Select promising actions S according to $policy(x)$;
3. If there is only one element in S, goto 8;
4. For $a \in S$ do
   4.a. Simulate action a; $(y,r) \leftarrow$ predicted new state and reinforcement;
   4.b. $E_a \leftarrow r + \gamma * eval(y)$;
5. $\mu \leftarrow \sum_{a \in S} Prob(a) \cdot E_a$; $max \leftarrow Max_{a \in S} E_a$;
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 2-5: The iteration of relaxation planning

The algorithm shown in Figure 2-5 is a modification to what is called *relaxation planning* in [Sutton 90]. In essence, with a world model, the evaluation and the policy networks can be effectively adjusted by looking ahead one step. Roughly speaking, 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 next state y:

$$eval(x) = Max_{a \in actions} r + \gamma \cdot eval(y) \qquad (6)$$

Thus, if a correct world model is available, the utility of any state can be more accurately estimated by looking ahead one step. During learning, the evaluation network is adjusted to reduce the difference between the two sides of the equation.

While the evaluation network is adjusted, the policy network can be updated as well. First we compute the average utility, μ, of a state x (Step 5) under the assumption that the current policy and the stochastic action selector will be used throughout the future. The *Prob* function is the one used to compute the probability distribution for selecting actions. In this work, it is Equation (4). Next, the policy network is modified to increase/decrease the merits of actions which are above/below the average (Step 7).

The algorithm of this framework is similar to that of Framework A, except that before the action selection step (Step 2 in Figure 2-2), relaxation planning may occur. To be efficient, relaxation planning is performed selectively (Steps 2 & 3 in Figure 2-5). 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 a lot. As learning proceeds, relaxation planning is performed less and less often.

2.4. Framework BM

Framework BM is Framework B plus a world model, much as Framework AM is Framework A plus a world model. The algorithm is similar to that of Framework B, except that in Framework BM actions can be taken with a world model as well as with the real world. Figure 2-6 shows the algorithm for using a world model. In essence, for situations where the best action is obvious, no action simulation is done

with the world model for efficiency reason. If there are several actions that seem to be promising, then these actions are tried with the model. The utility network is modified using a similar procedure as the one depicted in Figure 2-4.

---

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 element in S, goto 6;
4. For $a \in S$ do
   4.a. Simulate action a; $(y,r) \leftarrow$ predicted new state and reinforcement;
   4.b. $U'_a \leftarrow r + \gamma * Max_{k \in actions}\, util(y,k)$;
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 2-6: The algorithm for Framework BM

---

3. The Dynamic Environment

The dynamical environment is a 25x25 cell world. A sample environment is shown in Figure 3-1. There are four kinds of objects in the environment: the agent ("I"), food ("$"), enemies ("E"), and obstacles ("O"). The outside of the world is considered as 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 3-1, and fifteen pieces of food are randomly placed on unoccupied cells. On each move, the agent has four actions to choose from: walking up, down, left, or right to an adjacent cell. If the agent attempts to walk into obstacles, it will remain at the same position.

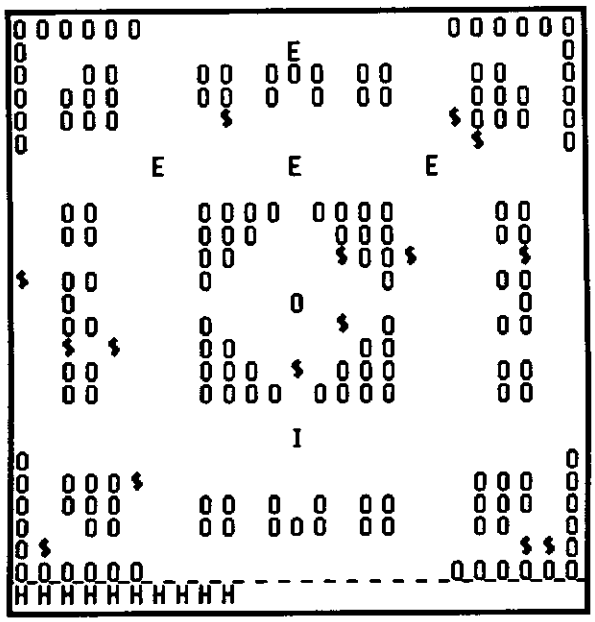

Figure 3-1: The dynamic environment

After the agent moves, each of the enemies is allowed to stay or move to an adjacent cell that is not occupied by obstacles. The speed of the enemies is about 80% of the full speed of the agent, so the agent is able to escape from chasing enemies. The behavior of the enemies is nondeterministic. However, there is a tendency that the enemies gradually move towards the agent, and the tendency becomes stronger when the agent is close.

The objective of the agent is to get as much food as possible while avoiding being caught by enemies. The play ends when the agent gets all of the food or dies. The agent can die by either colliding with the enemies or running out of energy. At the start of a new play, the agent is given 50 units of energy. Each piece of food provides the agent 15 units of energy, and each move costs the agent 1 unit. These three 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, 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 nondeterministically, but also because the world is only partially observable (so, what will be seen after a move is unpredictable). Although humans can avoid the enemies and get all of the food with little effort, survival in this environment is not trivial. To survive, an 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 for food when no food is within view.

4. The Learning Agents

In this section, I present the implementation of four learning agents, Agents A, B, AM, and BM, 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 modified using a symmetrical version of the error backpropagation algorithm [Stornetta & Huberman 87]. In other words, I use 0.5 for representing boolean true and -0.5 for boolean false, and the squashing function used is a variation of the sigmoid function:

$$f(x)=\frac{1}{1+e^{-x}}-\frac{1}{2} \qquad (7)$$

4.1. The Reinforcement Signals

After each move, the learning agent could receive from the environment one of the following reinforcement signals.

- -1.0 if the agent dies
- 0.4 if the agent gets food
- -0.05 if the agent moves back or walks into obstacles
- 0.0 otherwise

Negative reinforcement is considered bad, and positive is good. It is important to choose the reinforcement values, because they determine what an optimal agent's behavior will look like. Using too large a reinforcement value for having food may direct the agent to neglect enemies in the face of food, resulting in death, while using too small a reinforcement value may result in slow learning. I tried a few

different values for food reinforcement, and 0.4 gave roughly the best performance.

The agent gets a small penalty, -0.05, when it moves back to its previous position or attempts to walk into obstacles, because in general these actions are bad. In the earlier experiments, where this reinforcement was not used, the agent often moved back and forth when there were no interesting objects (i.e., food and enemies) around. The use of the reinforcement could instruct the agent to keep moving in a certain direction whenever no interesting objects are around, resulting in a behavior of seeking for food.

4.2. Input Representations

As described before, Agents A and AM consist of an evaluation network, a policy network, and a stochastic action selector, while Agents B and BM consist of a utility network and a stochastic action selector. The action selector is described in Section 2.1. This section describes the input representations of the three networks, while the next section describes the output representations.

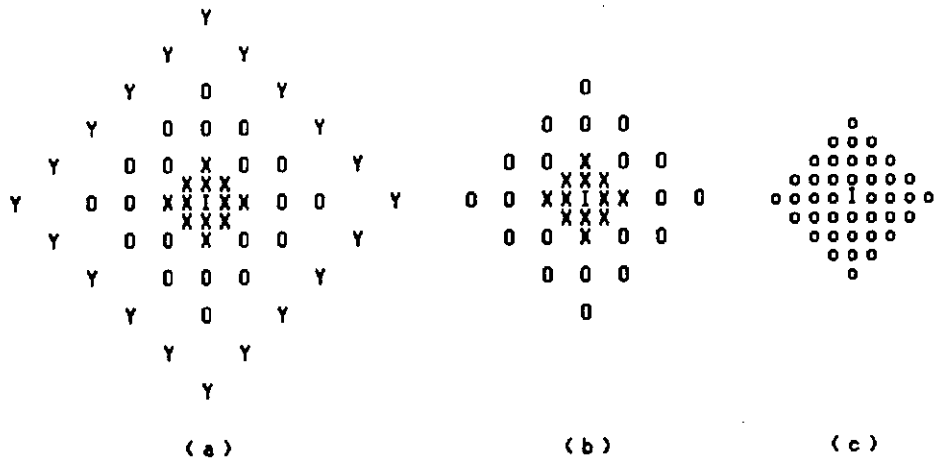

Figure 4-1: The (a) food, (b) enemy, and (c) obstacle sensor arrays

All of the three networks are 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 the agent's previous heading. Each of the maps shows a certain kind of objects in a local region surrounding the agent and can be thought of as being obtained by an array of sensors fixed on the agent. Figure 4-1 shows the configuration of the food, enemy and obstacle sensor arrays.

The spatial positions of food are coarse-coded [Rumelhart, et al. 86b], meaning that each food sensor may be activated by several nearby food objects. The food sensor array is composed of three different 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. 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 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 [Rumelhart, et al. 86b], and this is not the case for obstacles.

The agent's energy level is, again, coarse-coded by sixteen input units. In other words, 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, the networks use five units to represent the agent's previous heading (up, down, left, right, and stop). The heading information allows the agents to learn that moving back is generally bad.

4.3. Output Representations

Each of the three networks has one output unit, with a value between -1 and +1. (Of course, it must be scaled to a value between -0.5 and 0.5 before the backpropagation algorithm applies.) The output of the evaluation network represents the estimated utility of the given input state. The single output of the policy network represents the merit of moving "up". The merits of moving in other directions can be computed using the same network by rotating the input signals, except those for the energy level, by 90, 180, and 270 degrees. Similarly, the single output of the utility network represents the utility of moving "up" in response to the input state. Again, the utilities of moving in other directions can be computed by rotating the input signals appropriately. By taking advantage of the action symmetry, the number of needed hidden units is reduced, and the learning rate was found to be significantly improved.

4.4. The World Model

Both Agents AM and BM learn a world model. The world model is intended to model the input-output behavior of the dynamic environment. More specifically, given an action to be performed, the world 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 could have many possible outcomes. There are two alternatives to modeling the environment: the world 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 are quite easy to predict. So, to shorten simulation time without much simplying the model learning task, Agents AM and BM are only asked to learn a *reinforcement network* for predicting the reinforcement signal and an *enemy network* for predicting the enemy positions. The reinforcement network is structurally similar to the utility network of Agent B or BM. They differ in that the former estimates the reinforcement signal to be received immediately, while the latter estimates the discounted cumulative reinforcements to be received throughout the future.

The enemy network is also a three-layer, feed-forward network, consisting of 139 input units, 264 hidden units, and 99 output units. Unlike the other networks mentioned before, the enemy network is not fully connected and uses different inputs. Its inputs include the obstacle map and a high resolution, noncoarse-coded enemy map, which is more like the obstacle map than the enemy map mentioned before. (Note that the enemies' behavior is influenced by surrounding obstacles as well as the agent.) The outputs are also a high resolution enemy map, which is converted to a normal, coarse-coded enemy map by a hand-coded program. Since the enemies can only walk to an adjacent cell at each move, whether a pixel of the output enemy map will be activated is determined only by the nearby pixels of the input

enemy and obstacle maps. Therefore, the enemy network does not need to be fully connected. Instead, each of the output units is indirectly connected to (a small number of) nearby input units through 3 hidden units or so.

The reinforcement and enemy networks are learned on-line just like the other networks. However, learning the two networks is a kind of *supervised learning*, and the encountered experiences (i.e., input-output patterns) can be saved (for a certain period of time) and repeatedly presented to the networks. Roughly speaking, the world model stops improving after being trained on the experiences from playing the first 50 environments (roughly 5000 patterns). Because the enemies are nondeterministic, predictions by such a well-trained model sometimes can be very wrong. For example, a nearby enemy might unexpectedly appear or disappear in the predicted map.

In addition to a learned model, a "perfect" model is provided. The perfect model is in fact the environment simulator which simulates the enemies' random behavior. Agents AM and BM can use either the perfect model or a learned model, but not both. Next section will present a performance comparison between using a perfect model and using a learned, potentially incorrect one.

5. Evaluation

For each learning agent, there are several parameters that can be tuned for performance. These parameters include the number of hidden units (H), the learning rate (η) and momentum factor (α) of the backpropagation algorithm, and the temperature (T) of the stochastic action selector. The discount factor γ is fixed to be 0.9, however. The parameter values used to generate the results below were 1) T=0.02 (a pretty low temperature) for all of the agents, 2) H=40, η=0.2, and α=0.9 for the evaluation network, 3) H=30, η=0.4, and α=0.9 for the policy network, and 4) H=30, η=0.3, and α=0.9 for the utility network. These values were manually chosen to give roughly the best performance for Agents A and B. No search was done for Agents AM and BM; they simply used the values found for Agents A and B. I have tried cooling temperatures, but I found no advantage of using them.

To evaluate the four learning agents, 700 training environments and 10 testing environments were randomly generated. The agents were allowed to learn only from playing the 700 training environments. Each time an agent played 10 training environments, it was tested on the 10 testing environments. The average number of food pieces obtained by the agent in the testing environments was then plotted versus the number of training environments that the agent had played so far. The obtained learning curves were not smooth, primarily due to the enemies' random behavior. For ease of comprehension, the curves are smoothed by averaging every three successive points (the first two points of the curves are left untouched, however). Figure 5-1 show the smoothed learning curves of the agents. Agents AM and BM each have two learning curves. One is obtained from using a perfect model, and the other from using a learned, potentially incorrect model. Although the results shown below were obtained with a particular set of training and testing environments, I did another experiment with a different set of environments and obtained similar results.

To summarize these learning curves:

- All of the agents learned quickly at the early stage of learning. They reached 80% of the asymptotic performance after playing less than 100 environments.
- Before learning, the agents can only get about 0.3 out of 15 food pieces (2%). After playing 250 environments, Agent A can get 10 pieces (67%) in average, Agent AM 11.5 pieces (77%), and Agents B and BM 13 pieces (87%).

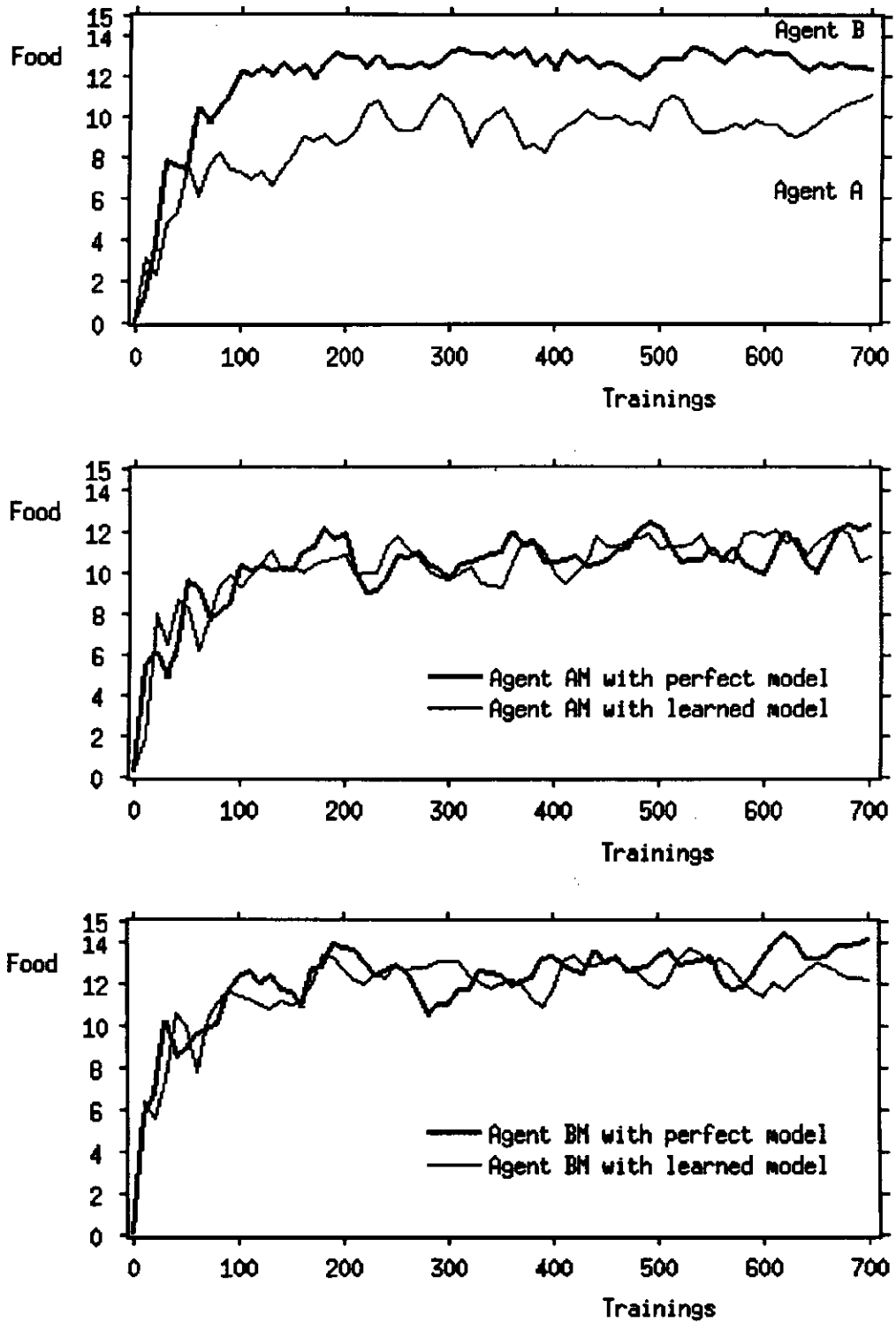

Figure 5-1: Learning curves of the agents

- If we sort the agents by the asymptotic performance, the ordering is roughly: BM (perfect model) ≈ B ≈ BM (learned model) > AM (perfect model) ≈ AM (learned model) > A.
- If we sort the agents by the average learning rate in the first 50 environments, the ordering is

roughly: BM (perfect model) ≈ BM (learned model) > AM (learned model) ≈ AM (perfect model) ≈ B > A.

Watching the agents playing, we can make the following observations:

- Agents B and BM were able to go get most food (including getting around obstacles) that were detected by the food sensors (the range of food sensors is 10 steps), while Agents A and AM were only able to go get food that is reachable within 6 and 8 steps or so, respectively.
- Agents B and BM learned good strategies for many rare and difficult situations, while Agent A did not. For instance, in the situations shown in Figure 5-2, Agent A would often move towards the food and get killed, while Agents B and BM would seldom make such mistakes. Agent AM was between Agents A and B.
- Even for handling common situations, Agents B and BM had better strategies than Agents A and AM. Agent AM was better than Agent A.
- Agents B and BM played very well. As a matter of fact, they got 14 or 15 pieces of food most of the time. They even learned how to get a collection of food efficiently, not simply take the closest one first. Of course they also made mistakes. Overlooking a potential danger is the most serious mistake. For example, being attracted by a piece of food, they might fall into a pocket formed by enemies and obstacles, and eventually get killed.

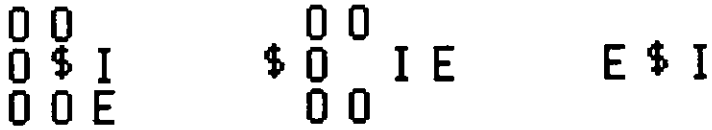

Figure 5-2: Difficult situations for Agent A

6. Discussion

Here is an interesting question: Why Agents B and BM performed better than Agents A and AM? My conjecture is as follows: Agents A and AM derive a policy from a learned evaluation function. In this fairly complex environment, the derivation process can be hardly perfect. On the other hand, Agents B and BM learn an evaluation function and use it directly for choosing actions. Thus, if all of the agents learn an evaluation function equally well, the performance of Agent A or AM would not be as good as that of Agent B or BM.

Another interesting question is why Agents B and BM performed equally well (in terms of asymptotic performance), while Agent AM was apparently better than Agent A? The reason is: What Agent BM gains from using a model is just additional experiences, while the relaxation planning provides Agent AM a more effective way to compute the evaluation function and to assess the relative merits of actions in response to the same situation-- this is not possible for Agent A to do. Because of this, we can expect that an agent using the model-based algorithm investigated in [Sutton 90] will be only as good as Agent A.

Not surprisingly, by using a world model, Agents AM and BM learned faster than their counterparts, A and B, even though the world model used was imperfect. However, if a sufficiently good world model is difficult to obtain, any advantage of using it could be negated. For example, Agent BM using a learned model performed slightly worse than Agent B.

In this work I only investigated how to use a world model to speed learning. 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, a world model, and a game-tree search algorithm to select actions. In a fairly complex domain, an optimal policy is often unobtainable. By looking ahead a few steps, the non-optimality of a policy can be compensated, if a good world model is available.

7. Conclusion

I have studied four reinforcement learning frameworks. Frameworks A and AM learn an evaluation function of world states and construct a policy to keep themselves in good states as measured by the evaluation function, while Frameworks B and BM learn only an evaluation function of state-action pairs and use it to choose actions. Frameworks AM and BM also learn a world model and use it as another source for having experiences. I also have developed four learning agents, based on the frameworks, to survive in a dynamic environment. Through the study of the learning agents, I now have better understanding about the learning frameworks.

In general Frameworks B and BM work better than Frameworks A and AM. By using a world model, Frameworks AM and BM can learn faster than their counterparts, A and B, in spite of an imperfect model used. Framework AM not only learns faster but also performs better than Framework A. However, even if a perfect model is used, Framework BM can be only as good as Framework B in terms of asymptotic performance.

While it seems that all of the learning frameworks are promising, a good implementation is also important to building a successful learning agent. In the course of developing the learning agents, three techniques were found to be very helpful: 1) the coarse coding technique, 2) the use of concentric multiple resolution maps, which dramatically reduces the number of input units needed, and 3) taking advantage of the action symmetry (i.e., treating the four move actions as one by rotating input signals appropriately). Without using these techniques, the agents could not have been so successful.

Possible future work includes 1) adding complexity to the environment (e.g., moving food and pushable obstacles) to see the limitation of the learning agents, 2) using an evaluation function, a world model, and game-tree search techniques to select actions, 3) investigating other learning frameworks, and 4) applying the methodologies to real world tasks.

8. Acknowledgements

I would like to express my gratitude to Tom Mitchell and Jeffrey Schlimmer for their inspiration and advice on this work.

References

[Anderson 89] Anderson, C.W.
Strategy learning with multilayer connectionist representations.
In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 103-114. 1989.

[Barto, Sutton & Watkins 89]
Barto, A.G., Sutton, R.S., and Watkins, C.J.C.H.
Learning and sequential decision making.
COINS Technical Report 89-95, Dept. of Computer and Information Science, University of Massachusetts, 1989.

[Kaelbling 89] Kaelbling, L.P.
A Formal Framework for Learning in Embedded Systems.
In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 350-353. 1989.

[Rumelhart, et al. 86a]
Rumelhart, D.E., Hinton, G.E., and Williams, R.J.
Learning internal representations by error propagation.
Rumelhart, D.E. and McClelland, J.L., editors, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1.
Bradford Books/MIT press, Cambridge, MA, 1986.

[Rumelhart, et al. 86b]
Hinton, G.E., McClelland, J.L., and Rumelhart, D.E.
Distributed representations.
Rumelhart, D.E. and McClelland, J.L., editors, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1.
Bradford Books/MIT press, Cambridge, MA, 1986.

[Stornetta & Huberman 87]
Stornetta, W.S., and Huberman, B.A.
An improved three-layer, back propagation algorithm.
In *Proceedings of IEEE International Conference on Neural Networks, vol.2*, pages 637-640. 1987.

[Sutton 84] Sutton, R.S.
Temporal Credit Assignment in Reinforcement Learning.
PhD thesis, Department of Computer and Information Science, University of Massachusetts, 1984.

[Sutton 88] Sutton, R.S.
Learning to predict by the methods of temporal differences.
In *Machine Learning, 3:9-44.* 1988.

[Sutton 90] Sutton, R.S.
First results with DYNA, an integrated architecture for learning, planning and reacting.
In *Stanford Spring Symposium on Planning.* 1990.

[Watkins 89] Watkins, C.J.C.H.
Learning with Delayed Rewards.
PhD thesis, Psychology Department, Cambridge University, 1989.

[Williams 87] Williams, R.J.
A class of gradient-estimating algorithms for reinforcement learning in neural networks.
In *Proceedings of the First Annual International Conference on Neural Networks, Vol II*, pages 601-608. 1987.