Programming Robots Using Reinforcement Learning and Teaching

Long-Ji Lin
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
e-mail: ljl@cs.cmu.edu

Abstract

Programming robots is a tedious task. So, there is growing interest in building robots which can learn by themselves. Self-improving, which involves trial and error, however, is often a slow process and could be hazardous in a hostile environment. By teaching robots how tasks can be achieved, learning time can be shortened and hazard can be minimized. This paper presents a general approach to making robots which can improve their performance from experiences as well as from being taught. Based on this proposed approach and other learning speedup techniques, a simulated learning robot was developed and could learn three moderately complex behaviors, which were then integrated in a subsumption style so that the robot could navigate and recharge itself. Interestingly, a real robot could actually use what was learned in the simulator to operate in the real world quite successfully.[1]

Introduction

Building learning robots is challenging, because it faces noises in robot sensing, uncertainties in robot actuators, and nondeterminism of the real world. Besides, fast convergence of learning is strongly demanded, simply because it is infeasible for a real robot to perform trial and error tens of thousands of times.

A variety of approaches to robot learning have been studied. A *supervised learning* approach has been used, for example, by Pomerleau [7] to control a vehicle to stay on roads using neural networks. In this approach, an external teacher is required to create training examples covering most of the situations which might be encountered on roads. The advantage of this approach is that learning can be done off-line and hence the robot would not suffer from any hazard during learning. The disadvantage is that without a teacher the robot cannot learn new strategies for situations not covered by the training examples.

Reinforcement learning is an *unsupervised learning* approach. In this approach, a learning agent receives from its environment a scalar performance feedback called *reinforcement* after each action. The agent's task is to construct a function (called *policy*) from states to actions so as to maximize the *discounted cumulative reinforcements* or *return* or *utility* [9, 12, 5]. Lin [5] for example, shows that several agents using different reinforcement learning algorithms learned to survive well in a dynamic environment, involving moving enemies and stationary food and obstacles. As opposed to supervised learning, the advantage of reinforcement learning is that the agent can improve performance from experiences without a teacher, but its main disadvantage is that learning converges slowly especially when the search space is large.

This paper proposes a technique (called *experience replay*) to speed up reinforcement learning and a general approach to combining reinforcement learning and teaching together. In this proposed approach, learning agents can take advantage of informative training examples provided by a teacher and speed up learning significantly, in the meanwhile, it can also self-improve through trial and error when a teacher is not available.

This paper also describes a simulated robot, which could learn three moderately complex behaviors; namely, wall following, door passing, and docking. The three behaviors were then integrated in a subsumption style [1, 6], enabling the robot to navigate in a simulated laboratory and corridor and to dock on a charger. This simulated robot and environment was intended to mimic a real robot called Hero [4] and a real world as closely as possible. Interestingly enough, the real robot could actually use what was learned in the simulator to operate in the real world quite successfully.

Reinforcement Learning and Teaching

Reinforcement learning often deals with two problems: the temporal and structural credit assignment problems. In this paper, Sutton's *Temporal Difference* (TD) methods [10] are used to solve the temporal credit assignment problem and the error backpropagation algorithm [8] to solve the structural one. More specifically,

[1]This work was supported partly by NASA under Contract NAGW-1175 and partly by Fujitsu Laboratories Ltd.

Q-learning [12], a direct application of TD methods, is employed to learn a function, $Q(x,a)$, which maps from state-action pairs, (x,a), to the expected returns. The Q-function is implemented using neural networks, one for each action. Each network's inputs are the state, and its single output represents the expected return from the execution of the corresponding action in response to the input state. While TD methods compute the errors in the current Q-function, the backpropagation algorithm adjusts the networks to reduce the errors. Lin [5] presents a detailed study of various reinforcement learning algorithms, including Q-learning.

Experience Replay

There are two inefficiencies in the 'traditional' Q-learning which Watkins proposed in his thesis [12]. First, experiences collected by trial and error are used to adjust the Q-function only once and then thrown away. This is wasteful, because some experiences are costly to obtain and some are rare. Experiences should be reused in a more effective way. Second, presenting a sequence of experiences to the Q-learning algorithm in chronologically forward order is less effective in propagating credits back through the sequence than presenting it in backward order, even when multiple-step Q-learning [12] is used.

To remove the two inefficiencies, I propose a technique called *experience replay*. Before describing the technique, I would like to be more specific about two terms I use a lot in this paper. An *experience* is a quadruple, (x_t, a_t, x_{t+1}, r_t), meaning that at time t action a_t in response to state x_t results in the next state x_{t+1} and reinforcement r_t. A *lesson* is a sequence of experiences starting from an initial state x_0 to a final state x_n where the goal is achieved or given up. By experience replay, the agent remembers a lesson and repeatedly presents the experiences in the lesson in **chronologically backward** order to the algorithm depicted in Figure 1, where γ, $0 \leq \gamma < 1$, is a *discount factor* and λ, $0 \leq \lambda \leq 1$, is the recency parameter used in TD(λ) methods [10].

The idea behind this algorithm is as follows. Roughly speaking, the expected return (called TD(λ) return) from (x_t, a_t) can be written recursively [12] as

$$R(x_t, a_t) = r_t + \gamma[(1-\lambda)U(x_{t+1}) + \lambda R(x_{t+1}, a_{t+1})] \quad (1)$$

where

$$U(x_t) = Max\{Q(x_t, k) \mid k \in actions\} \quad (2)$$

Note that the term in the brackets [ ] of Equation 1, which is the expected return from time $t+1$, is a weighted sum of 1) return predicted by the current Q-function and 2) return actually received in the replayed lesson. With $\lambda = 1$ and backward replay, $R(x_t, a_t)$ is exactly the discounted cumulative reinforcements from time t to the end of the lesson. In the beginning of learning, the Q-function is usually far from correct, so the actual return is often a better estimate of the expected return than that predicted by the Q-function. But when the Q-function becomes more and more accurate, the Q-function provides a better prediction. In particular, when the replayed lesson is far back in the past and involves bad choices of actions, the actual return (in that lesson) would be smaller than what can be received using the current policy (In this case, it is better to use $\lambda = 0$). Thus, to estimate returns, it would be better to start with $\lambda = 1$ in the beginning of learning and then gradually reduce λ to zero as learning proceeds.

To replay $\{(x_0, a_0, x_1, r_0)...(x_n, a_n, x_{n+1}, r_n)\}$, do

1. $t \leftarrow n$
2. $e_t \leftarrow Q(x_t, a_t)$
3. $u_{t+1} \leftarrow Max\{Q(x_{t+1}, k) \mid k \in actions\}$
4. $e'_t \leftarrow r_t + \gamma[(1-\lambda)u_{t+1} + \lambda e'_{t+1}]$
5. Adjust the network implementing $Q(x_t, a_t)$ by backpropagating the error $e'_t - e_t$ through it
6. if $t = 0$ exit; else $t \leftarrow t - 1$; go to 2

Figure 1: Algorithm for experience replay and teaching

Ideally, we want $Q(x_t, a_t) = R(x_t, a_t)$ to hold true. The difference between the two sides of "=" is the error between *temporally successive predictions* about the expected return from (x_t, a_t). To reduce this error, the networks which implement the Q-function are adjusted using the backpropagation algorithm (Step 5).

With $\lambda = 0$, this algorithm is just the one-step Q-learning. As discussed above, for faster credit propagation, it would be better to use a decreasing value of λ. However, for simplicity, in all the experiments of this work, λ was fixed to be 0.3, and set to 0 when (1) $t = n$ because e'_{t+1} in Step 4 will be undefined, or (2) action a_{t+1} is not a current policy action (see the discussion above and [12]). (Later, I will discuss how to decide whether an action is a current policy action or not.)

It is instructive to relate experience replay to what Sutton called *relaxation planning* [11, 5]. By relaxation planning, the learning agent uses a learned action model, a function from (x_t, a_t) to (x_{t+1}, r_t), to (randomly) generate hypothetical experiences for updating the Q-function. Experience replay in fact is a kind of relaxation planning. Instead of using a learned action model, experience replay uses the collection of past experiences as the "model". This "model" represents not only explicitly the environment input-output behavior but also the probability distribution of multiple outcomes of actions. Experience replay is more effective than the kind of relaxation planning used in Sutton's Dyna architecture in two ways: (1) because of backward replay and use of nonzero λ value, credit propagation should be faster, and (2) there is no need to learn a model, which sometimes is a difficult task [5].

Teaching

It is helpful to learn from a teacher, because a teacher may provide informative training examples which are difficult to obtain by trial and error methods. Teaching can be easily integrated with reinforcement learning. In fact, the same algorithm for experience replay can be employed to learn from a teacher. Teaching can be conducted in the following way. First, a teacher shows how a target task can be achieved from some sample initial state. The sequence of the shown actions as well as the state transitions and received reinforcements is recorded as a *taught lesson*. Several taught lessons can be collected and replayed by the learner just in the same way *experienced* (i.e., self-generated) lessons are replayed.

Unlike supervised learning, this approach to teaching does not require the teacher to show only optimal solutions to the robot. However, if a teacher can provide optimal solutions to some sample tasks, learning might converge most quickly by using $\lambda = 1$ when the taught lessons are replayed.

The main problem with replaying either experienced or taught lessons is that harmful over-training of the networks might occur when some experiences are presented to the learning algorithm too many times, because the networks may become too specific to those experiences. A partial solution is discussed later.

A Robot and Its Tasks

In this work, a real robot called Hero [4] is used to study robot learning. The robot has a sonar sensor and a light sensor mounted on its top. Both sensors can rotate and collect 24 readings (separated by 15 degrees) per cycle. The sonar sensor returns a distance reading between 0 and 127 inches, with a resolution of 0.5 inch. A simple thresholding to the light readings can detect a light source with a directional error less than 15 degrees. The robot can move forward and backward, and can turn to any direction. But to use the learning algorithm, motions must be discretized. (This is a limitation!) The robot is allowed to execute 16 different actions. It may turn 15, 30, 45, 60, 75, 90, 180, $-15, -30, -45, -60, -75$, or -90 degrees and then move 10 inches forward. Or it may move 10 or 20 inches forward, or 10 inches backward.

The robot's task is to move around in our lab and corridor, which are connected by a narrow door, and to dock on a charger when requested. Learning this task as a whole is difficult, because there is a limitation on TD methods for which learning is practical. For example, to find and dock on a charger may need 40 or more robot actions. Credit propagation through such a long sequence of actions would be slow and requires the Q-function to be represented to high precision, which is already traded off for generalization by connectionist implementation. One way of getting around this limitation is to use *hierarchical control*; in other words, decompose the whole task into subtasks, learn each subtask separately, and combine what are learned for each subtask together in some way. In their work on a box-pushing robot, Mahadevan and Connell [6] show that a behavior-based robot which learns each component behavior separately is better, in terms of performance and learning speed, than a robot which attempts to learn the box-pushing task as a whole.

For the above reason, the task of the Hero robot is decomposed into three smaller tasks, each of which the robot will learn a behavior to carry out. The three tasks are (1) following walls and obstacles, (2) going through the door from any location near the door, and (3) docking on the charger from any location near the charger. The ideas behind this decomposition are as follows: Both docking and door-passing require the robot to position itself relative to the charger or the door within small errors. To do both tasks from a distant location, the robot can first follow walls until it is near the door or the charger, and then activate a specialized docking or door-passing behavior to position itself and go for it.

Using a real robot to collect experimental data, however, is very time-consuming; each sensing operation of the Hero robot, for instance, takes 15 seconds or so. Therefore, a simulator was developed as a substitute, which was intended to mimic the real robot and real world as closely as possible, including simulating the errors in sensing and actions. Figure 2 shows the simulated robot environment.

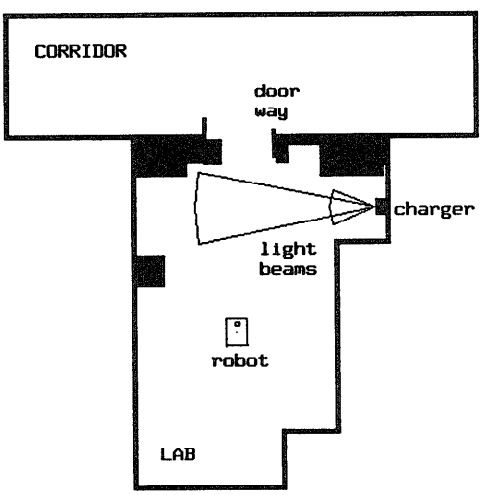

Figure 2: The simulated robot world

To enable the robot to find the charger, a light is put on the top of the charger. The docking task is nontrivial. Following the light beams does not necessarily end up success. Sonar data must also be used to help position the robot with errors within $\pm$ 4 inches in position and $\pm$ 12 degrees in orientation. Although door-passing

demands less precision than docking, sonar sensors are so noisy and unreliable in detecting the door opening. For instance, sonar may be absorbed by the soft part of chairs or reflected by smooth surfaces, resulting in door-like openings. This property of sonar sensors is also modeled in the simulator.

The Learning Robot

The Reinforcement Functions

The reinforcement signals used for docking are:
- 1.0 if docking succeeds
- −0.5 if collision occurs
- 0.0 otherwise.

The reinforcement signals used for door-passing are:
- 1.0 if door-passing succeeds
- 0.5 if door-passing succeeds but collision also occurs
- −0.5 if collision occurs
- 0.0 otherwise.

The reinforcement signals used for wall-following are:
- −0.5 if collision occurs
- −0.1 if robot is too far from or too close to walls
- 0.2 if action executed is "move 20 inches forward"
- 0.0 if action executed is "move 10 inches backward"
- 0.08 otherwise.

When the robot is about to follow walls, it first decides which side of walls to follow (usually the closest one) and keeps following walls on that side. The robot is requested to keep from obstacles on that side a distance between 20 and 45 inches. When the desired distance is kept, the robot receives a reward of 0.2, 0.08 or 0.0, depending on how much progress is made by the action executed.

Input Representations

As discussed before, the Q-function for each of the tasks is implemented using neural networks, one for each action. All of the networks used in this work are 3-layer, feed-forward, fully-connected networks (with no connections between input and output layers). The number of hidden units of each network is always 35% of the number of input units. Each unit uses a symmetric squashing function, which is the sigmoid function shifted down by 0.5. The learning rate and momentum factor used in the backpropagation algorithm are always 0.3 and 0.9, respectively. Each network has only one output unit, representing the expected return.

Each of the networks for the wall-following task consists of 25 input units, among which 24 units encode the 24 sonar readings, and 1 unit indicates whether a collision is detected or not. Three different ways of encoding sonar data have been tried. They all gave good performance. Among them, the best one is to encode readings ranging from 0 to 64 as values between 1 and 0, and to encode readings ranging from 64 to 127 as values between 0 and $-1/4$. Small readings are more weighted because the robot has to pay more attention to nearby obstacles than to far obstacles. In this work, all sonar readings are encoded in this way, while all binary inputs are represented as 0.5 and −0.5.

For the docking task, each network consists of 50 input units — 24 units for sonar readings, 24 units for (binary) light readings, 1 unit for the collision feature, and 1 unit for indicating if a light is seen in the previous state. The last unit encodes a kind of temporal information, which may or may not be useful. It is the robot's task to figure out how to use this piece of information. Similarly, for the door-passing task, each network consists of 50 input units — 24 units for sonar readings, 24 units for door-opening features, 1 unit for the collision feature, and 1 unit for indicating if a door-opening is detected in the previous state. Much like the light readings, the door-opening features, which are made out of the sonar readings, indicate the direction of any door-like opening. As mentioned before, the door-opening features are only clues and very unreliable.

Other Issues

To actively explore the consequences of different actions to similar situations, the learning robot, during learning, chooses actions stochastically according to a Boltzmann distribution:

$$Prob(a_i) = exp(Q(x, a_i)/T)/\sum_k exp(Q(x, a_k)/T) \quad (3)$$

where T (called *temperature*) adjusts the randomness of action selection. A cooling temperature is used; it starts with 0.05 and gradually cools down to 0.02 after 60 lessons have been collected. It turned out that using this strategy alone did not give a very good compromise between acting to gain information and acting to gain rewards. A complementary strategy is also used: the robot dead-reckons its trajectory and increases the temperature whenever it finds itself stuck in a small area without progress.

Attention must be paid to prevent harmful overtraining of networks when lessons are replayed. Replaying some experiences too many times may make the networks too specific to these particular experiences, resulting in poor generalization. My solution to this problem is to perform a screen test before an experience is to be replayed: If the probability (see Equation 3) of choosing the action according to the **current** Q-function is either higher than $P_u = 99.9\%$ or lower than $P_l = 0.1\%$, the experience is not replayed, because the robot already knows the action is either much better than others or much worse than the best one. Without taking this test, experience replay was found to be beneficial only in the beginning and then become harmful after a while.

As discussed before, λ is set to 0 when an action to be replayed is not a policy action. An action is considered as a non-policy action, if the probability of choosing that action (according to the current policy) is lower than $P_l = 0.1\%$.

Experimental Results

The experimental data presented below were obtained with the simulator. To collect experimental data for each task, three experiments were conducted under the same condition, except that different numbers of taught lessons were available to the robot. (To generate a taught lesson, I placed the robot in a random state and did my best to move the robot to the goal state.) Each experiment consisted of 300 trials. In each trial, the robot starts with a random position and orientation. For the wall-following task, each trial lasts for 15 steps. For the docking and door-passing tasks, each trial ends either when the goal is achieved, or else after 30 steps, whichever comes first. To measure performance over time, after every 10 trials, the robot is tested with a fixed set of 50 different initial positions and orientations. For the wall-following task, the average reinforcement obtained in a test is recorded, while for the docking and door-passing tasks, the average number of steps taken in a test is recorded. The measured performance is plotted versus the number of trials that have been performed.

The three experiments for each task were repeated five times with different random seeds. Figure 3 shows the typical learning curves, which are already smoothed, for the ease of comprehension, by averaging every 3 successive original data points (The first two points are left untouched.) The following are the observations from the experimental data:

1. For docking: Without a teacher, the robot was unable to learn the docking task. To dock, the robot must drive to collide with the charger for a tight connection, but this is contradictory to obstacle avoidance the robot would learn in the beginning. In fact, other researchers [2] have also found that reinforcement learning alone is unable to learn a good policy in a complex domain; it often converges to bad local maxima. On the other hand, with some teaching (even with only a few training instances), learning converged to considerably better performance.

2. For door-passing: Learning this task without a teacher sometimes converged as quickly as learning with a teacher, but sometimes it took a long time to converge. (The learning of connectionist networks is ill-characterized!) Eventually the robot could learn the task well with or without a teacher.

3. For wall-following: Compared with the other two tasks, this task is an easier one, because rewards/blames are not far delayed. Learning this task without a teacher usually converged almost as quickly as learning with a teacher, but sometimes converged to poor performance.

In general, learning with more taught lessons converged more quickly and/or to better performance, but this was not guaranteed. For instance, a sub-optimal taught lesson might lead the robot to learn a sub-optimal policy, although the robot might find the optimal policy after (considerable) exploration. Note that the lessons I taught the robot were not all the optimal solutions. In fact, it was difficult for me to tell the best action given a set of sensory readings. Monitoring the robot's learning process, I observed that the robot considered some of the taught actions as bad choices, even when the robot performed the tasks very well.

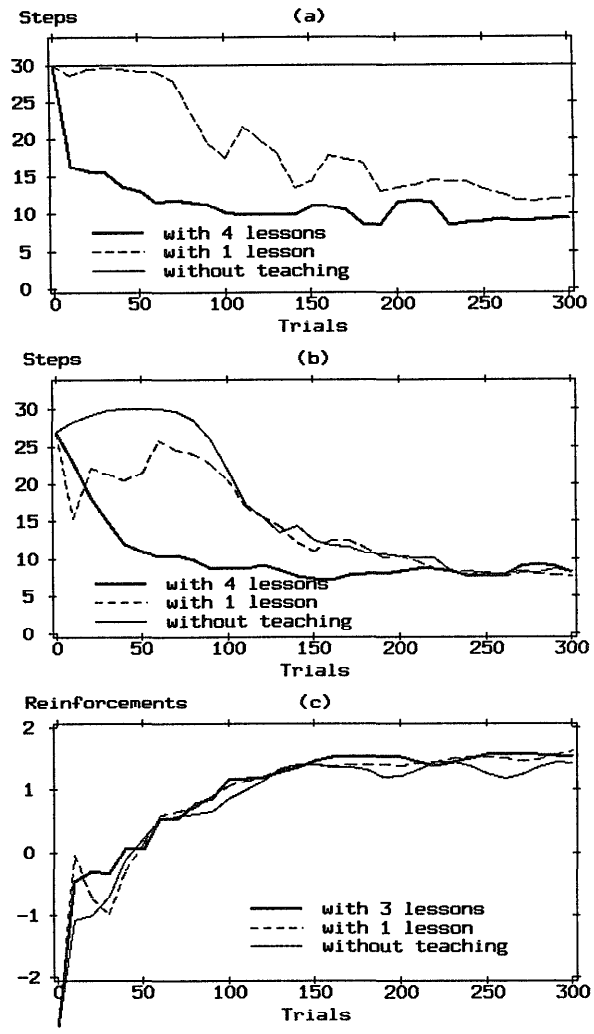

Figure 3: Learning curves for a) docking, b) door passing & c) wall following

Integration of Behaviors

As shown already, with some teaching, the robot learned to pass a narrow door from a place near the door. What if the robot starts with a position far from the door? In this case, the robot can first activate the wall-following behavior, which is then subsumed by the door-passing behavior when the door is nearby. To identify whether the door is nearby or not, however, is

a nontrivial pattern recognition task. While useful, the door-opening features alone are not sufficient in determining a nearby door. The Q-function for door-passing can help — if the value of $U(x)$ (see Equation 2) is high, the door should be reachable within some small number of steps. But the Q-function alone is also not sufficient in determining a nearby door. The condition used in this work for the door-passing behavior to subsume the wall-following behavior is a combined use of the door-opening features and the Q-function: (*"seeing a door-like opening"* and $U(x) > 0.2$) or ($U(x) > 0.5$). A similar condition was used to integrate the docking and wall-following behaviors. 0.2 and 0.5 in the condition were chosen empirically. A challenge would be letting the robot itself learn to coordinate multiple behaviors.

The integrated robot was also tested in both the simulator and the real world. For each test, a random goal (either docking or door-passing) is generated and the robot starts with a random position. In the simulator, 100 tests were performed, and 96% (more or less) of the time, goals were achieved within 50 steps. In the real world (where people walked around once in a while), 25 tests were performed, and 84% of the time, goals were achieved within 60 steps.

Related Work

Sutton [9, 10, 11] developed TD methods, the Adaptive Heuristic Critic (AHC) architecture, and relaxation planning. Q-learning was proposed by Watkins [12]. Lin [5] describes a detailed study of several learning algorithms, using AHC-learning, Q-learning and neural networks. Kaelbling [3] describes an *interval estimation* algorithm to control the tradeoff between acting to gain information and acting to gain reinforcements, but it is unclear how the algorithm can be used with connectionist approach.

Pomerleau [7] describes work on using neural networks to control the motion of an autonomous vehicle, but his approach does not allow self-improvement. Mahadevan and Connell [6] describe a box-pushing robot using a similar approach as mine, but they do not exploit teaching and re-using past experiences.

Conclusion

This paper presented an experience replay algorithm, which can speed up credit propagation significantly and can be used to integrate teaching with reinforcement learning. Reinforcement learning without teaching often converges to bad local maxima in a complex domain such as the docking task. With some teaching, learning can be considerably better off. This paper also demonstrated that reinforcement learning and teaching together is a promising approach to autonomous learning of robot behaviors. Since a Q-function indicates some kind of information about the applicability of a behavior (i.e., the larger the Q-value, the closer the goal), the integration of behaviors can benefit from the use of learned Q-functions.

Acknowledgements

I thank Tom Mitchell for many helpful discussions.

References

[1] Brooks, R.A. (1986). A Robust Layered Control System for a Mobile Robot. In *IEEE Journal of Robots and Automation, vol. RA-2, no. 1*

[2] Chapman, D. and Kaelbling, L.P. (1990). *Learning from Delayed Reinforcement In a Complex Domain.* Tech. Report TR-90-11, Teleos Research.

[3] Kaelbling, L.P. (1990). *Learning in Embedded Systems.* Ph.D. diss., Dept. of Computer Science, Stanford University.

[4] Lin, L.J., Mitchell, T.M., Phillips, A., and Simmons, R. (1989). *A Case Study in Autonomous Robot Behavior.* Tech. Report CMU-RI-89-1, Carnegie Mellon University.

[5] Lin, L.J. (1990). Self-improving Reactive Agents: Case Studies of Reinforcement Learning Frameworks. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats, 297-305.* Also Tech. Report CMU-CS-90-109, Carnegie Mellon University.

[6] Mahadevan, S. and Connell, J. (1990). *Automatic Programming of Behavior-based Robots using Reinforcement Learning.* IBM Research Report RC 16359, IBM Watson Research Center.

[7] Pomerleau, D.A. (1989). ALVINN: An Autonomous Land Vehicle in a Neural Network. Tech Report CMU-CS-89-107, Carnegie Mellon Univ.

[8] Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1,* Bradford Books/MIT press.

[9] Sutton, R.S. (1984). *Temporal Credit Assignment in Reinforcement Learning.* Ph.D. diss., Dept. of Computer and Information Science, University of Massachusetts.

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

[11] Sutton, R.S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning, 216-224.*

[12] Watkins, C.J.C.H. (1989) *Learning with Delayed Rewards.* Ph.D. diss., Psychology Department, Cambridge University.