Reinforcement Learning in Non-Markov Environments

Steven D. Whitehead
GTE Laboratories Inc.,
40 Sylvan Road
Waltham, MA 02254

Long Ji Lin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

October 13, 1992

Abstract

Recently, techniques based on reinforcement learning (RL) have been used to build systems that learn to perform non-trivial sequential decision tasks. To date, most of this work has focussed on learning tasks that can be described as Markov decision processes (MDPs). While MDPs are useful for modeling a wide range of control problems, there are important problems that are inherently non-Markov. We refer to these as *hidden state* tasks since they arise when information relevant to identifying the state of the environment is *hidden* (or missing) from the agent's internal representation. Two important types of control problems that resist Markov modeling are those in which 1) the system has a high degree of control over the information collected by its sensors (e.g., as in active-vision), or 2) the system has a limited set of sensors that do not always provide adequate information about the current state of the environment. Not surprisingly, traditional RL algorithms, which are based primarily upon the principles of MDPs, perform unreliably on hidden state tasks.

This article examines several approaches to extending RL to hidden state tasks. A generalized technique called the *Consistent Representation (CR) Method* is described. This method unifies such recent approaches as the Lion algorithm, the G-algorithm, and CS-QL; however it is restricted to a class of problems which we call *adaptive perception tasks*. Several, more general, *memory-based* algorithms that are not subject to this restriction are also presented. Memory-based algorithms, though quite different in detail, share the common feature that each derives its internal representation by combining immediate sensory inputs with internal state which is maintained over time. The relative merits of all of these methods are considered and conditions for their useful application are given.

1 Introduction

This article is concerned with techniques for building systems that learn control. We are specifically interested in sequential control tasks. These are tasks in which control unfolds over time through a series of control actions generated by an autonomous control system. In sequential control, the controller (or agent), in choosing a control action, must take into account not only an action's immediate effect, but also its impact on future states. Examples of sequential control range from the very simple (e.g., pole balancing) to the very complex (e.g., human behavior).

We are interested in learning for several reasons. First, we are interested in systems that can adapt to changing conditions and changing tasks. A system whose behavior is completely determined ahead of time is less useful than one that can learn a new task or adapt to changes in the environment. Also, learning can simplify design by relieving the developer from the burden of specifying a full controller. Instead of deriving an optimal controller by carefully analyzing a domain *a priori* (an impossible job in some cases), it may be more efficient to install an initially suboptimal controller, which through learning is optimized.

In this article we focus on the reinforcement learning paradigm. The central concept underlying reinforcement learning is to formulate control tasks as optimization problems by providing the system with state dependent payoffs (rewards and penalties). Under this scenario, the objective of the system is to learn a state-dependent control policy that maximizes a measure of payoff received over time. Early examples of reinforcement learning include Minsky's maze running automata [26], Samuel's checker player [32], and Michie and Chamber's pole balancer [25]. Other examples include Sutton's Adaptive Heuristic Critic [37, 9], Sutton's Temporal Difference Methods [38], Holland's Bucket Brigade [17], and Watkins Q-learning [47]. More recent work addresses a wide range of issues including modularity [12, 29, 24, 35, 57, 54], incremental planning [39, 49, 19], efficient credit assignment [59, 19], intelligent exploration [18, 44], efficient representations [27] and neural implementations [55, 34, 19].

With respect to sequential control, attention has traditionally focussed on learning to control Markov decision processes. Described formally in Section 2, a Markov decision process intuitively corresponds to a control task in which at each point in time the controller has a description (or representation) of the external environment which specifies all information relevant for optimal decision-making. This total information assumption is called the *Markov* assumption and is important for two reasons. First, the Markov assumption has been important to the theoretical development of RL [47, 48, 38], since focusing on Markov decision processes has allowed researchers to apply the classical mathematics of stochastic processes and dynamic programming. Second, existing reinforcement learning methods depend upon the Markov assumption for credit assignment and often perform badly when the assumption is violated. Nevertheless, there are important control problems that are not naturally (or easily) formulated as Markov decision processes. These non-Markov tasks are commonly referred to as *hidden state* tasks, since the occur whenever it is possible for a relevant piece of information to be *hidden* (or missing)

from the controller's representation of the current situation.

Hidden state tasks arise naturally in the context of autonomous learning robots. For example, if a robot's internal representation is defined solely (or largely) by its immediate sensor readings, and if there are circumstances in which the sensors do not provide all the information needed to uniquely identify the state of the environment with respect to the task, then the decision problem facing the embedded controller is non-Markov. Hidden state tasks are also a natural consequence of active/selective perception [7, 6, 4]. In active perception, the agent has a degree of control over the allocation of its sensory resources (e.g., controlling visual attention or selecting of visual processing modules). This control is used to sense the environment in an efficient, task-specific way. However, if control is not properly maintained then the data generated by the sensors may say nothing important about the current state of the environment and the agent's internal representation will be ambiguous. It follows that if the agent must learn to control its sensors there will be periods of time in which the internal representation will be inadequate. Therefore the decision-task will be non-Markov.

Techniques for applying reinforcement learning to non-Markov decision processes is the central focus of this article. We describe a generalized technique called the *Consistent Representation (CR) Method* that can be used to learn control in systems with active perception [50]. The principal idea underlying the CR-method is to split control into two phases, a perceptual phase and an overt phase. During the perceptual phase, the system performs sensing (or sensor configuration) actions in order to generate an adequate (read Markov) representation of the current external state. During the overt stage, this representation is used to select overt action; that is, actions that change the state of the external environment. Systems using the consistent representation method learn not only the overt actions needed to perform a task, but also the perceptual actions needed to construct an adequate, task-specific representation of the environment. The CR-method unifies such recent algorithms as the Lion Algorithm [52], CS-QL [41], and the G-algorithm [13]. However, it is limited to tasks in which the agent can identify, at each point in time, and through proper control of its sensors, the current state of the environment with respect to the task. We refer to this limited class of tasks as *adaptive perception tasks*.

Several, more general, *memory-based* algorithms that are not restricted to involving adaptive perception are also described. The simplest approach augments the systems sensory inputs with a delay line to achieve a crude form of short term memory [21]. This approach has been successful in certain speech recognition tasks [46]. Another alternative is the method of predictive distinctions [14, 21, 33, 5] Following this approach the system learns to predict sensory inputs (or environmental observables) and then uses the internal state of the predictive model to drive action selection. A third approach uses a recurrent neural network in combination with classical reinforcement learning methods to learn a state dependent control policy (and utility function) directly [21]. Each of these methods is described and analyzed in detail.

The remainder of the article is organized as follows. Section 2 provides a basic

review of concept from reinforcement learning and the Theory of Markov decision processes. Section 3 discusses sources of non-Markov processes and considers the difficulties they cause for traditional reinforcement learning methods. Section 4 presents the Consistent Representation Method as a technique for dealing with adaptive perception and reviews examples of this technique. Section 5 considers the delay line method, the predictive distinctions method and the use of recurrent neural networks for dealing with hidden state tasks, in general. Comparisons are drawn between approaches and preference conditions are specified for each. Section 6 discusses all of these methods in the broader context of scalability and conclusions are drawn in Section 7.

2 Review of Elementary Concepts

Before getting into the details of non-Markov decision problems, the consistent representation method, and techniques for dealing with hidden state, it is useful to establish context by reviewing some of the models and techniques that are most prevalent in the reinforcement learning literature. To this end, we now turn to a brief description of a model of agent-environment interaction that is widely used in reinforcement learning. We also review Markov decision processes and Q-learning [47], an algorithm popular in the reinforcement learning community. Unfortunately, a thorough review of Markov decision processes and reinforcement learning in general is beyond the scope of this article. Therefore, we focus primarily on Q-learning and the difficulties caused for it by non-Markov decision processes. Other algorithms found in the literature [9, 17, 55, 38] suffer a similar fate. For a more complete review of Markov decision processes and Q-learning, the reader may wish to consult [11] and [47]. For a review of reinforcement learning in general see [8].

2.1 Modeling agent-environment interaction

Figure 1 illustrates a model of agent-environment interaction that is widely used in reinforcement learning research. In this model the agent and the environment are represented by two synchronized finite state automatons interacting in a discrete time cyclical process. At each point in time, the following series of events occur.

1. The agent's sensory system measures properties of the environment and constructs a description of the current state of the environment.
2. Based on this internal representation, the agent's embedded controller chooses a motor command to perform.
3. The motor-command is transformed through the agent's motor interface into action in the world.
4. Based on the action issued by the agent and the current state, the environment makes a transition to a new state and generates a reward.
5. The reward is passed back to the agent.

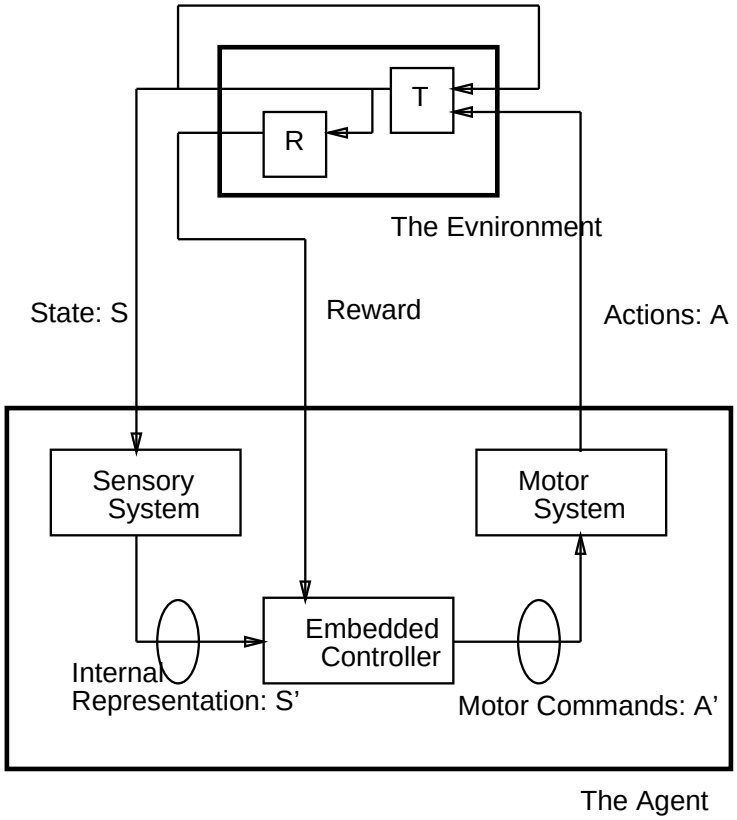


Figure 1: A simple model of agent-environment interaction. The environment is modeled as a discrete time, discrete state Markov decision process. The agent can directly sense the state of the environment (i.e. the state of the process) and its actions map directly to the process model's actions.

2.1.1 The Environment

The automaton representing the environment is modeled as a Markov decision process (also called a controllable Markov process). Formally, a Markov decision process is described by the tuple (S, A, T, R), where S is a set of possible states, A is a set of possible actions, T is a state transition function, and R is a reward function. The environment at each time point occupies exactly one state from S, and accepts at each point a single action from A. S and A are usually assumed to be discrete and finite. The dynamics of state transitions are modeled by a *transition function*, T, which maps state-action pairs into new states ($T : S \times A \rightarrow S$). The transition function is generally probabilistic. That is, $X_{t+1} = T(x_t, a_t)$, where X_{t+1} is the random variable denoting the state at time $t + 1$, and x_t and a_t denote the state and action performed at time t, respectively. T is typically specified in terms of a set of *transition probabilities*, $P_{x,y}(a)$, where

$$P_{x,y}(a) = Prob(T(x, a) = y). \tag{1}$$

The rewards generated by the environment are determined by a *reward function*, R, which maps states into scalar-valued payoffs (rewards/penalties) ($R : S \rightarrow \Re$). In general, the reward function is also probabilistic, such that $R_t = R(x_t)$, where R_t is the random variable denoting the reward received at time t.

Notice that in a MDP, the effects of actions (in terms of the next state and immediate reward received) only depend upon the current state. Process models of this type are said to be memoryless and satisfy the *Markov property*. The Markov property is fundamental to this model of the environment since it implies that knowledge of the current state is precisely the information needed for optimal control (that is for maximizing the reward generated over time). Thus, even though it may be possible to devise action-selection strategies that base their decisions upon additional observations about the environment, these strategies cannot possibly outperform the best decision strategies that depend only upon knowledge of the current state!

2.1.2 The Agent

The agent automaton consists of three components a sensory interface, a motor interface and an embedded controller. The sensory interface implements a mapping from the set of external states S to the set of possible internal representations S'. The motor interface implements a mapping from internal motor commands A' to external actions A. The embedded controller is responsible for generating control actions. At each time step, it receives input from the sensors and generates an action command that is interpreted by the motor interface. The controller also receives rewards generated by the environment. These rewards are used as feedback for learning. In many cases, the sensory-motor interface between the controller and the environment is not explicitly modeled.[1] In this case, the embedded controller is

[1]This is possible since many of these experiments occur in simulation where the boundary between the "external world" and the agent's internal representation are easily blurred.

given direct access to the state of the environment, and issues actions directly to the world. When the sensory system is explicitly modeled (or implemented), it typically corresponds to a fixed set of sensors that are carefully chosen to provide precisely the information relevant to control.[2] Under these circumstances, the mapping from internal states to external states is *functional* and the decision problem facing the embedded controller is Markov. *Perceptual aliasing* is said to occur when the mapping from the internal state space, S', to the external state space, S, is *not functional*[50]. In this case, a non-Markov Decision process obtains. We shall talk about perceptual aliasing at length in Section 3. However, for the remainder of this section, we shall neglect the sensory-motor interface and assume the controller accesses S and A directly.

2.1.3 Policies and the objective of control

One way to specify an agent's behavior is in terms of a control *policy*, which prescribes, for each state, an action to perform. Formally, a policy f is a function from states to actions ($f : S \rightarrow A$), where $f(x)$ denotes the action to be performed in state x.

In reinforcement learning, the agent's objective is to learn a control policy that maximizes some measure of the total reward accumulated over time. In principle, any number of reward measures can be used, however, the most prevalent measure is one based on a discounted sum of the reward received over time. This sum is called the *return* and is defined for time t as

$$return(t) = \sum_{n=0}^{\infty} \gamma^n r_{t+n} \tag{2}$$

where γ, called the temporal discount factor, is a constant between 0 and 1, and r_{t+n} is the reward received at time $t + n$. Because the process may be stochastic, the agent's objective is to find a policy the maximizes the *expected return*.

For a fixed policy f, define $V_f(x)$, the *value function* for f, to be the expected return, given that the process begins in state x and follows policy f thereafter. The agent's objective is to find a policy, f^*, that for every state maximizes the value function. That is, find f^*, such that

$$V_{f^*}(x) = \max_f V_f(x) \qquad \forall x \in S. \tag{3}$$

An important property of MDPs is that f^* is well defined and guaranteed to exist. In particular, the *Optimality Theorem* from dynamic programming [10] guarantees that for a discrete time, discrete state Markov decision process there always exists a deterministic policy that is optimal. Furthermore, a policy f is optimal if and only if it satisfies the following relationship:

$$Q_f(x, f(x)) = \max_{a \in A}(Q_f(x, a)) \qquad \forall x \in S. \tag{4}$$

[2] Motor commands almost always have a one-to-one mapping to the actions in a Markov model of the task.

Q_f, is called the *action-value function*, and $Q_f(x,a)$ is defined to be the return the agent expects to receive given that it starts in state x, applies action a next, and then follows policy f thereafter [10, 11]. Intuitively, Equation 4 says that a policy is optimal if and only if in each state, the policy specifies the action that maximizes the local "action-value." That is,

$$f^* = \arg\max_{a \in A}[Q_{f^*}(x,a)] \qquad \forall x \in S, \tag{5}$$

and

$$V_{f^*}(x) = \max_{a \in A}[Q_{f^*}(x,a)] \qquad \forall x \in S. \tag{6}$$

For a given MDP, the set of action-values for which Equation 4 holds is unique. These values are said to define the optimal action-value function Q^* for the MDP.

If an MDP is completely specified *a priori* (including the transition probabilities and reward distributions) then dynamic programming techniques can be used to compute an optimal policy directly [10, 30, 11]. However, because we are interested in learning, we assume that only the state space S and set of possible actions A are known *a priori* and that the statistics governing T and R are *unknown*. Under these circumstances the agent cannot compute the optimal policy directly, but must explore its environment and learn an effective control policy by trial-and-error.

2.2 Q-learning

Q-learning is an incremental reinforcement learning method [47]. It is a good representative of reinforcement learning because it is simple, elegant, mathematically well founded, and widely used. For our purposes Q-learning is useful for illustrating the difficulties caused by non-Markov decision problems. Also, because other reinforcement learning algorithms use similar credit assignment techniques (namely TD-methods [38]), an understanding of the difficulties caused by non-Markov decision problems for Q-learning goes a long way toward understanding weaknesses of other algorithms [9, 17, 56, 34]. For a detailed treatment of Q-learning see [47].

In Q-learning the agent estimates the optimal action-value function directly, and then uses it to derive a control policy using the local greedy strategy mandated by Equation 5. A simple Q-learning algorithm is shown in Figure 2. The first step of the algorithm is to initialize the agent's action-value function, Q. Q is the agent's *estimate* of the optimal action-value function. If prior knowledge about the task is available, that information may be encoded in the initial values, otherwise the initial values can be arbitrary (e.g., uniformly zero). Next the agent's initial control policy, f, is established. This is done by assigning to $f(x)$ the action that locally maximizes the action-value. That is,

$$f(x) \leftarrow \arg\max_{a \in A}[Q(x,a)]. \tag{7}$$

Ties are assumed to be broken arbitrarily. After initialization, the agent enters the main control/learning cycle. First, the agent senses the current state, x. It then selects an action a to perform next. Most of the time, this action will be

$Q \leftarrow$ a set of initial values for the action-value function (e.g., uniformly zero)
For each $x \in S$: $f(x) \leftarrow a$ such that $Q(x,a) = \max_{b \in \mathbf{A}} Q(x,b)$,
Repeat forever:

1) $x \leftarrow$ the current state
2) Select an action a to execute that is usually consistent with f but occasionally an alternate. For example, one might choose to follow f with probability p and choose a random action otherwise.
3) Execute action a, and let y be the next state and r be the reward received.
4) Update $Q(x,a)$, the action-value estimate for the state-action pair (x,a):
   $Q(x,a) \leftarrow (1-\alpha)Q(x,a) + \alpha[r + \gamma U(y)]$
   where $U(y) = Q(y, f(y))$.
5) Update the policy f:
   $f(x) \leftarrow \arg\max_{a \in A}[Q(x,a)]$

Figure 2: A simple version of the 1-step Q-learning algorithm.

the action specified by its policy $f(x)$, but occasionally the agent will choose a random action.[3] The agent executes the selected action and notes the immediate reward r and the resulting state y. The action-value estimate for state action pair (x,a) is then updated. In particular, an estimate for $Q^*(x,a)$ is obtained by combining the immediate reward r with a utility estimate for the next state, $U(y) = \max_{b \in A}[Q(y,b)]$. The sum

$$r + \gamma U(y), \tag{8}$$

called a 1-step corrected estimator, is an unbiased estimator for $Q^*(x,a)$ when $Q = Q^*$, since, by definition

$$Q^*(x,a) = E[R(x,a) + \gamma V^*(T(x,a))], \tag{9}$$

where $V^*(x) = \max_{a \in A} Q^*(x,a)$. The 1-step estimate is combined with the old estimate for $Q(x,a)$ using a weighted sum:

$$Q(x,a) \leftarrow (1-\alpha)Q(x,a) + \alpha[r + \gamma U(y)], \tag{10}$$

where α is the learning rate. Finally, the agent's control policy is updated using Equation 5, and the cycle repeats. If, in the limit, every state-action pair is tried

[3]Occasionally choosing an action at random is a particularly simple mechanism for exploring the environment. Exploration is necessary to guarantee that the agent will eventually learn an optimal policy. For examples of more sophisticated exploration strategies see [18, 44, 39].

infinitely often and if the learning rate is decreases according to a proper schedule, Q-learning is guaranteed to converge to an optimal policy for any finite Markov decision processes [48].

2.3 An Example

As a simple example, consider the maze task illustrated in Figure 3. In this problem, the agent, "A," is free to roam about a bounded 2-dimensional maze. It can move in one of four principle directions, left, right, up, or down, but it cannot pass through barriers. Actions are deterministic. The agent has longitude and latitude sensors that accurately locate its position in the maze. The task is to navigate to the cell labeled "G". To entice the agent to the goal, a small positive reward is generated each time it enters the goal cell. In other states, the agent receives no reward. To facilitate exploration, upon entering the goal cell the agent is teleported to a new random location in the maze. In this way, the temporal evolution of the process resembles a sequence of repeated trials.

The state space for this task is determined by the possible values for the location sensors. In this case, the maze size is a 10×10, so there are a total of 100 distinct cells, 75 which are not occupied by a barrier. With a choice of four possible actions per state, the agent must estimate a total $75 \times 4 = 300$ action-values. If the agent is initially ignorant of the underlying structure of the environment and the position of the reward, then it cannot accurately estimate the optimal action-value function. Under these circumstances a particularly simple approach is to initialize all action-values to zero. In this case, the agent's initial performance will be random (assuming ties are broken by choosing randomly), and useful change in the action-value function first occurs when the agent first encounters the goal state. At that point, the action-value for the state-action pair that immediately proceeded the goal state is increased. On subsequent trials, the action-values of other state-action pairs are incrementally increased as they are found to lead to either reward states or states of high utility. In this way, reward information is "backed up" until an accurate estimate of the optimal action-value function is obtained and an optimal control policy is learned. Figures 3b) and c) show examples of policies learned after 10 and 100 trials, respectively (for $\alpha = 1.0$). After 10 trials, reward information has been propagated to only 11 states; whereas, after 100 trials action-values in nearly every state have been effected by the reward generated at the goal.

3 Non-Markov Tasks

The model depicted in Figure 1 is widely used in the reinforcement learning literature and has been usefully applied to a number of simple adaptive control problems. A key assumption made by the model is that the decision problem facing the embedded controller is Markov. However, there are important tasks that are not naturally formulated according to this model, and that lead more naturally to non-Markov models. In particular, a non-Markov decision tasks arise any time it is possible

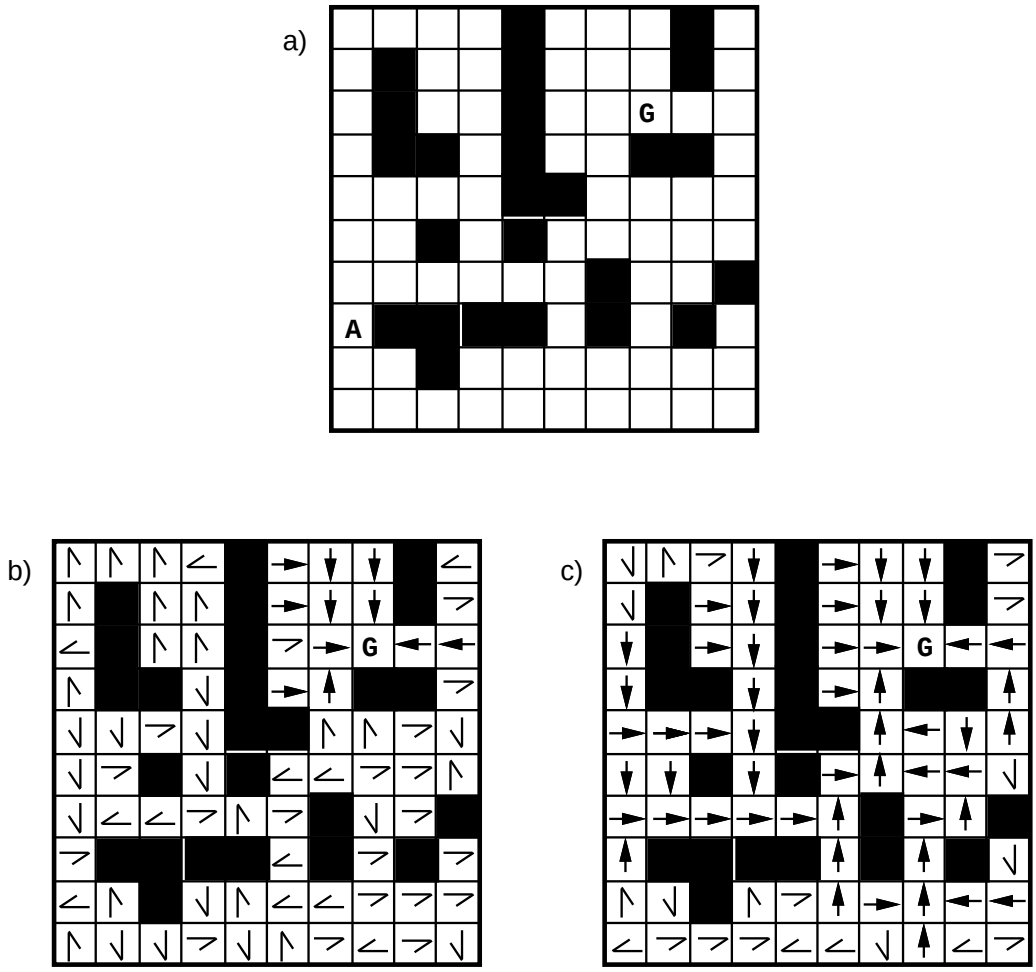


Figure 3: A simple reinforcement learning task: a) the maze; b) a policy learned after 10 trials; c) a policy after 100 trials. Filled arrows indicate cells whose action-values have been effected by reward from the goal state.

for the controller to be uncertain about the state of the external environment. In this section, we describe two ways in which non-Markov decision problems arise in the context of autonomous robot learning. We also describe the problems caused by non-Markov decision problems for Q-learning, in particular, and reinforcement learning, in general.

3.1 Active Perception

Active perception refers to the idea that an intelligent agent should actively control its sensors in order to sense and represent only the information that is relevant to its immediate ongoing activity. That a system's sensors should be matched to its intended control task is clear, and for relatively simple control tasks a set of fixed, matched sensors may be adequate. However, active perception differentiates itself from other approaches to perceptual organization when the agent's behavior is scaled to include a variety of complex tasks that come and go over time. In this case the body of information relevant at any point in time changes as different phases of the task unfolds and as the agent moves from one task to another. For example, the items of immediate interest to a boy on the playground varies depending upon whether he is playing tag, kickball, baseball or soccer, and on whether he is at bat, on second, or playing Centerfield. Under such diverse and time dependent information needs, an active perception paradigm (and efficiency considerations) mandate an active approach to sensing. Human vision is an example of active perception. People efficiently move their eyes to foveate objects of behavioral significance and register peripheral objects with much lower resolution [58]. Similarly, work aimed at developing vision systems for robots has recently seen a shift toward active sensing [7, 2, 45, 6, 4]

Active perception is relevant to adaptive control and reinforcement learning for two reasons. First, agents equipped with active sensory systems pose interesting and important adaptive control problems. In particular, given a robot with an active sensory-motor system, can we build a controller, based on reinforcement learning (or other techniques) that effectively learns to control both perception and action. Second, adaptive control coupled with active sensing provides an opportunity to overcome a limitation of our previous model (Figure 1), namely, that right from the beginning the agent's sensory system generates an internal representation that uniquely identifies the state to the environment at each point in time. This assumption implies that the designer of the system knows enough about the task ahead of time to identify the relevant state variables and to choose sensors (and sensing procedures) that measure them efficiently. In a learning system with active perception, it may be possible to equip the system with a very flexible sensory system and have it learn to extract relevant bit of information (i.e., a dynamic, task-specific representation) as it simultaneously learns control.

Tasks that involve active perception lead naturally to non-Markov decision problems since improper control of the sensors leads to internal representations that fail to encode information relevant for decision making. This point is illustrated schematically in Figure 4 which shows the basic structure of a system with active

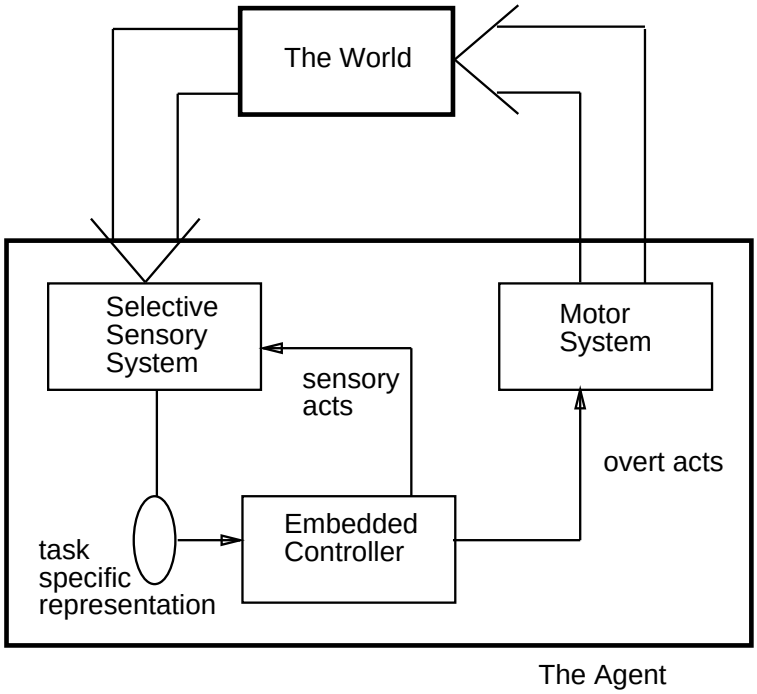


Figure 4: A simple model of the structure of a system with active perception.

perception. The key feature of the diagram is the arrow that feeds from the embedded controller back to the sensory system. It represents control information (or commands) used to modulate the sensory mapping. In this case, the sensory system can be thought of as implementing a series of sensory mappings, from states in the external model to bits in the internal representation. The control signals are used to select (or modify) a sensory mapping. Since the controller does not initially know an appropriate sensing strategy it may, during the course learning, adopt sensory mappings that neglect relevant information, a phenomenon we call *perceptual aliasing*. Formally perceptual aliasing occurs whenever a state in the internal representation maps to (or represents) two or more states in the external Markov model of the task. Perceptual aliasing causes the embedded decision problem to be non-Markov since, under these circumstances, it is impossible to define a set of state-based transition and reward probabilities (over the internal state space) that accurately capture the dynamics of the external process. We define the class of *adaptive perception tasks* to be those control problems in which the agent has an active sensory system, which when properly configured can uniquely identify each and every state in a Markov model of the external task. Moreover, we shall assume that the sensory systems used in these tasks are such that changing the configuration of the sensory system in no way effects the state of the external environment.

3.2 Other kinds of hidden state tasks

Of course active perception is not the source of non-Markov decision problems. A non-Markov task arises whenever relevant information is missing from the agent's internal representation. If a situated system depends solely (or largely) upon its immediate sensory inputs for decision making, then if for any reason a relevant piece of information is hidden from its sensors the resultant control problem is non-Markov. Lin [23] provides a good example:

> Consider a packing task which involves 4 steps: open a box, put a gift into it, close it, and seal it. An agent driven only by its current visual precepts cannot accomplish this task, because when facing a closed box the agent does not know if the gift is already in the box and therefore cannot decide whether to seal or open the box.

In this case occlusion of the gift by the lid prevents immediate perception of a vital piece of information. Hidden state tasks also arise when temporal features (such as velocity and acceleration) are important for optimal control, but not included in the system's primitive sensor set.

3.3 Effects on Learning

The straightforward application of reinforcement learning to non-Markov decision problems in almost all cases fails to yield an optimal policy, and in most cases results in bad performance. Perceptual aliasing and the non-Markov decision problems that result from it cause two problems for existing reinforcement learning. First, like the dynamic programming methods on which they are based, most existing reinforcement learning algorithms aim to learn an optimal policy that is deterministic. However, unlike Markov processes, which are guaranteed to have a deterministic optimal policy, the optimal policies for non-Markov processes are frequently non-deterministic.[4] Second, learning algorithms like Q-learning [47], AHC [37], and Bucket Brigade [17] adapt their control policies by maximizing a local evaluation function (e.g., the action-value function in Q-learning, and a state-based utility function in AHC). However, for non-Markov decision problems accurate estimates for local evaluation functions cannot be obtained for states that are perceptually aliased. This leads to localized errors in the policy function. Moreover, use of temporal difference methods [38] for temporal credit assignment spreads estimation errors throughout the state space, thus infecting even policy actions for non-aliased states.

To illustrate these problem more concretely, let us examine the effects of applying Q-learning to a simple non-Markov decision problem. Consider the task shown in Figure 5. In this task, the external decision problem has a state space containing eight states, $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, g\}$; two actions, $A_E = \{l, r\}$; and a deterministic transition function, shown in Figure 5a. The goal of the external task is to

[4] That is instead of associating a single action with each state, a optimal policy involves associating with each state a (non-trivial) probability distribution over possible actions.

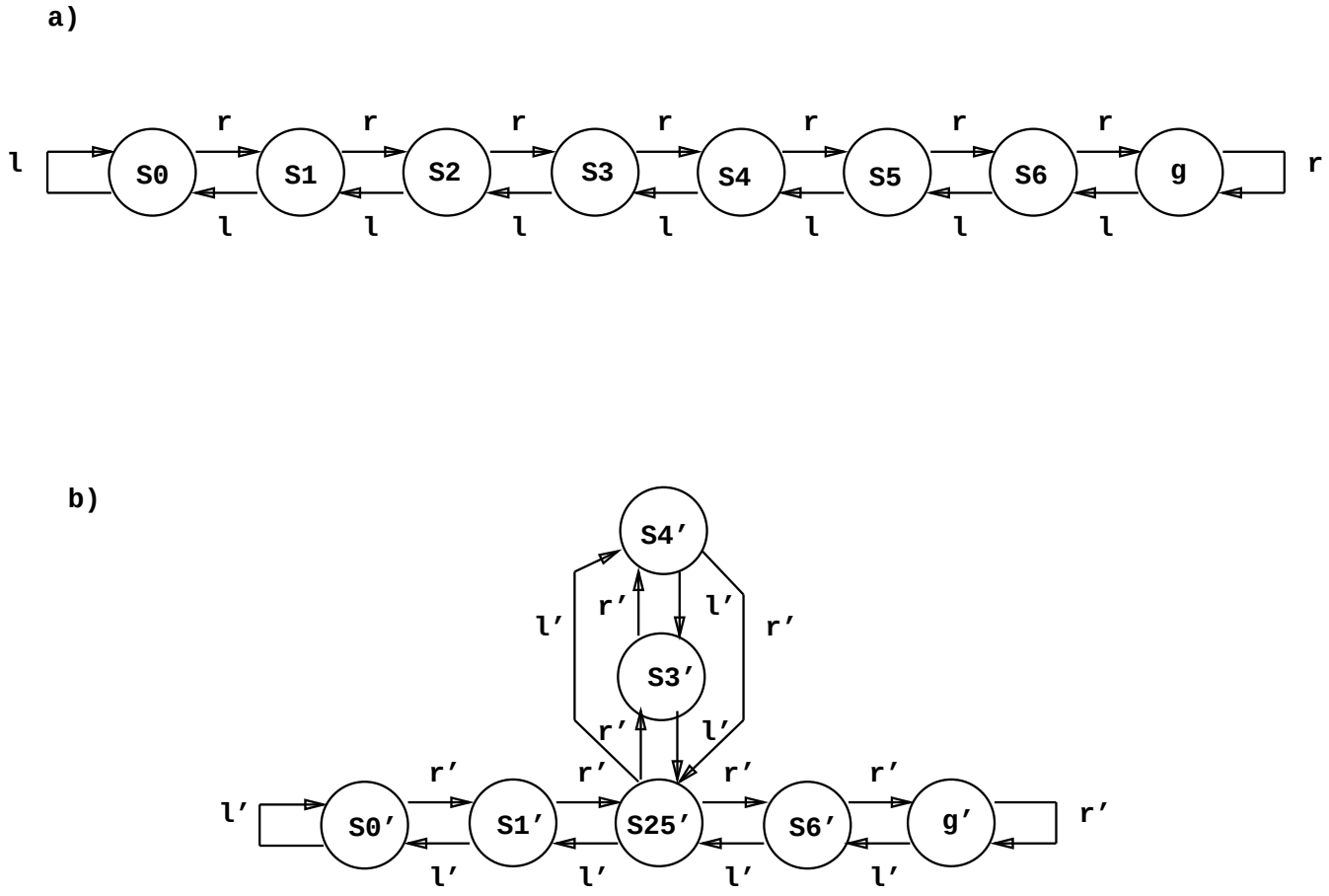


Figure 5: An illustration of the difficulties caused by perceptual aliasing. Transition diagrams for a simple decision task: a) the transition diagram for the external decision problem, b) the transition diagram for the internal (or perceived) decision problem when interpreted through a sensory-motor system with perceptual aliasing.

enter the goal state g, whereupon the agent receives a fixed reward $R(g) = 5000$. Non-goal states yield zero reward, $R(s_k) = 0$ for $k = 0$ to 6.

The optimal value function for this external Markov task, denoted V_E^*, is an exponentially decreasing function of the distance to the goal. That is, $V_E^*(s) = R_E(g)\gamma^{(d(s)-1)}$, where $d(s)$ is the distance (in steps) from state s to the goal. The optimal policy, f_E^*, corresponds to choosing the action that minimizes the distance to the goal. In this case, the optimal policy requires the agent to moving right (r) at every opportunity (i.e., for all $s \in S$, $f_E^*(s) = r$). Notice that the optimal solution path for a given trial traces out a trajectory where $V_E^*(x_t)$ is monotonically increasing in time, and that the optimal policy corresponds to performing a gradient ascent of V_E^*. This result is illustrated in Figure 6a, which plots $V_E^*(x_t)$ versus time for a trial that begins in state s_0 at time $t = 0$ and follows the optimal trajectory to g at time $t = 7$. When applied directly to this problem, the Q-learning algorithm described in Figure 2 can easily learn the optimal policy. However, let us introduce perceptual aliasing into the sensory mapping and see what happens.

Consider the internal decision problem that results when the agent's sensory-motor system implements a perceptual mapping that is fixed, one-to-one, and onto except for states s_2 and s_5, which gets mapped onto the same internal state, $s'_{2,5}$. That is, let $S' = \{s'_0, s'_1, s'_{2,5}, s'_3, s'_4, s'_6, g'\}$, where except for $s'_{2,5}$, s'_j (and g') represents world state s_j (and g). Also let the motor mapping be such that $A' = \{l', r'\}$, where l' and r' map to l and r, respectively. The transition diagram for this internal decision problem is shown in Figure 5b. Notice that this decision problem is non-Markov since the effects of actions are not independent of the past but depend upon the hidden, unperceived external state (namely when the internal state reads $s'_{2,5}$). Also note that a fixed optimal policy for this task is to always apply the action r'.

1-step Q-learning cannot learn the optimal policy for this task. In particular, when the agent's policy is initialized to the optimal policy and the controller is fixed so that the system follows the optimal policy with probability $p = 0.99$ and chooses a random action otherwise, and the system is run for a long series trials, which would otherwise be adequate to learn the optimal value and action-value functions, the following is observed. First, since the value and action-value estimates (U_I and Q_I respectively) are based on *expected* returns, for the state $s'_{2,5}$, they take on values somewhere between the corresponding values for s_2 and s_5 in the external decision problem. That is,

$$V_E^*(s_2) \leq U_I(s'_{2,5}) \leq V_E^*(s_5), \tag{11}$$

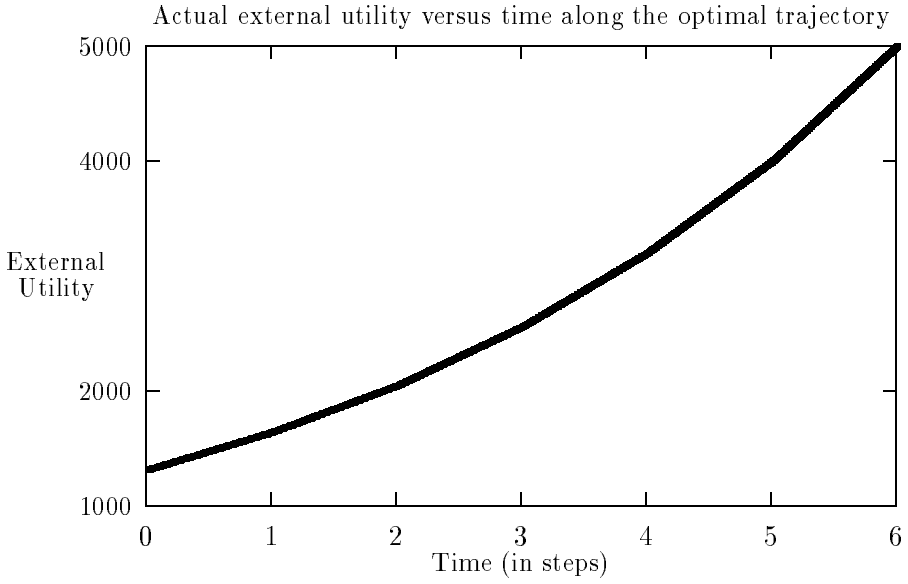
$$Q_E^*(s_2, r) \leq Q_I(s'_{2,5}, r') \leq Q_E^*(s_5, r), \tag{12}$$

and

$$Q_E^*(s_2, l) \leq Q_I(s'_{2,5}, l') \leq Q_E^*(s_5, l). \tag{13}$$

In fact, the estimated action-value function does not even converge to the true sampled average of the returns observed. This follows since to update its action-value function, the agent uses a 1-step estimator which enforces only local constraints on the values estimated. If the learning rate is gradually decreased with time, the

a)



b)

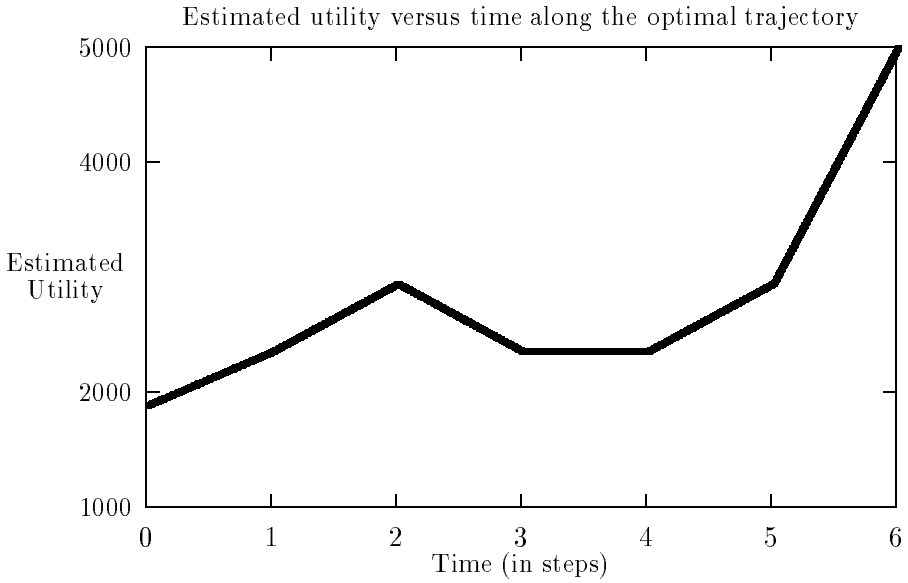


Figure 6: Plots of utility versus time as the agent traverses from state s_0 at $t = 0$ to g at $t = 7$ (for $\gamma = 0.8$): a) the utility for the external decision problem, V_E^*; b) the utility estimates for the internal decision problem, U_I, obtained by the 1-step Q-learning algorithm.

	s'_0	s'_1	s'_3	s'_4	$s'_{2,5}$	s'_6
$U_I(s)$	1882	2352	2352	2352	2941	5000
$Q_I(s,l')$	1506	1506	2352	1882	1882	2352
$Q_I(s,r')$	1882	2352	1882	2352	2941	5000
$V_S(s)$	1310	1638	2560	3200	3024	5000
$Q_S(s,l')$	1048	1048	1310	1638	1935	3200
$Q_S(s,r')$	1310	1638	2560	3200	3024	5000

Table 1: The utility and action-value functions estimated by the 1-step Q-learning algorithm and the true sampled utility and action-value functions. The estimated functions do not match the true sampled values since they are obtained by satisfying the local constraints imposed by the corrected 1-step estimator. The estimated utility and action-values are denoted U_I and Q_I, respectively, and the sampled utility and action-values are denoted V_S and Q_S. The values shown are for $\gamma = 0.8$.

action-value function estimated by the agent converges to the values that satisfy the following local relationships:

$$Q_I(s'_6, r') = R(g') + \gamma 0 = 5000 \tag{14}$$
$$Q_I(s'_{2,5}, r') = f_1[0 + \gamma U_I(s'_6)] + f_2[0 + \gamma U_I(s'_3)] \tag{15}$$
$$Q_I(s'_4, r') = 0 + \gamma U_I(s'_{2,5}) \tag{16}$$
$$Q_I(s'_3, r') = 0 + \gamma U_I(s'_4) \tag{17}$$
$$Q_I(s'_1, r') = 0 + \gamma U_I(s'_{2,5}) \tag{18}$$
$$Q_I(s_0, r') = 0 + \gamma U_I(s'_1) \tag{19}$$
$$Q_I(s'_6, l') = 0 + \gamma U_I(s'_{2,5}) \tag{20}$$
$$Q_I(s'_{2,5}, l') = f'_1[0 + \gamma U_I(s'_4)] + f'_2[0 + \gamma U_I(s'_1)] \tag{21}$$
$$Q_I(s'_4, l') = 0 + \gamma U_I(s'_3) \tag{22}$$
$$Q_I(s'_3, l') = 0 + \gamma U_I(s'_{2,5}) \tag{23}$$
$$Q_I(s'_1, l') = 0 + \gamma U_I(s'_0) \tag{24}$$
$$Q_I(s'_0, l') = 0 + \gamma U_I(s'_0) \tag{25}$$

where $U_I(x) = \max_{a \in \{l', r'\}} Q_I(x, a)$.

In Equation 15, f_1 and f_2 are the fraction of times the application of r' in state $s'_{2,5}$ results in the next states being s'_3 and s'_6, respectively. Similarly, in Equation 21, f'_1 and f'_2 are the fraction of times the application of l' in state $s'_{2,5}$ results in states s'_1 and s'_4, respectively. If trials always begin in state s'_0, then $f_1 = f_2 = f'_1 = f'_2 = 50\%$ and the values for the utility and action-value functions will converge on the values shown in Table 1. Also shown in the table are the sampled utility and action-values (V_S and Q_S, respectively), obtained by actually measuring and averaging the returns received over many trials (instead of using a 1-step estimator). Notice that the sampled averages match the optimal values for

the external decision task except for states s_2 and s_5. In this case,

$$V_S(s'_{2,5}) = 1/2V^*_E(s_2) + 1/2V^*_E(s_5) \tag{26}$$

$$Q_S(s'_{2,5}, l') = 1/2Q^*_E(s_2, l) + 1/2Q^*_E(s_5, l) \tag{27}$$

$$Q_S(s'_{2,5}, r') = 1/2Q^*_E(s_2, r) + 1/2Q^*_E(s_5, r). \tag{28}$$

Notice that the utility and action-values estimated by Q-learning, except for state s_6, do not match either the external or the sampled utility and action values. This discrepancy arises because estimates for all the states up to s_5 (i.e., $s'_0, s'_1, s'_{2,5}, s'_3$, and s'_4) in the internal task are either directly or indirectly dependent upon the utility estimate for s_5. However, since s_2 and s_5 are indistinguishable, their internal action-value estimates are constrained to be the same, and consequently inaccurate. These inaccurate utility estimates in turn get propagated back to the states in the state space.

Another observation to make is that the utility function (either learned or measured) for the internal decision problem no longer increases monotonically as the system traverses the optimal solution trajectory. This anomaly is shown graphically in Figure 6b, which plots $U_I(x_t)$ as a function of time as the system follows the optimal trajectory from s'_0 to g'. The plot shows that a utility aberration occurs at $t = 2$ when the system first encounters $s'_{2,5}$. At this point, the environment is in state s_2 and the true expected return is $V^*_E(s_2) = 2048$ (for $\gamma = 0.8$). However, because s_2 and s_5 are indistinguishable in the internal representation, the internal decision system overestimates the expected return at $t = 2$. Similarly, another estimation error occurs the second time $s'_{2,5}$ is encountered, at $t = 5$ when the environment is in state s_5. In this case, $U_I(s'_{2,5})$ underestimates the expected return.

If we relax our hold on the decision policy and allow the system to adapt, we find that the optimal policy is unstable! Not only is the system unable to find the optimal policy, it actually moves away from it. In general, the system will oscillate among policies, never finding a stable one. The instability can be understood by considering the effect of utility estimation errors on the policy. Recall that in Q-learning the system locally adjusts its policy in order to maximize the expected return. Thus, after running the agent with a fixed policy for many trials and then releasing it, the policy value for state s'_3 will be changed so that the system tends to take actions that move it back to $s'_{2,5}$ instead of forward to s'_4 (since $Q_I(s'_3, l') > Q_I(s'_3, r')$). The large utility value for state $s'_{2,5}$ acts as an attractor for nearby states, such as s'_3, and causes them to change their local policy away from optimal. An intuitive way to understand the problem is to consider a local homunculus that sits at s'_3 and can see the utilities of its neighbors. From his point of view, $s'_{2,5}$ looks more desirable than s'_4 since once the system is in $s'_{2,5}$ it can execute r', which often leads to s'_6 (one step from the goal). On the other hand, choosing the action which leads to s'_4 leaves the system still three steps from the goal. From the homunculus' point of view, going to $s'_{2,5}$ is on average better than going to s'_4. What the homunculus cannot perceive (because of perceptual aliasing) is that going from s'_3 directly to $s'_{2,5}$ always returns the real external world

to state s_2, which cannot reach s_6 directly. The problem is that the homunculus cannot distinguish between s_2 and s_5, as they are both represented by $s'_{2,5}$, and it erroneously makes the Markov assumption — that the effects of actions depend only upon the current perceived state.

Errors in the utility function are also unstable since they are based on a running average of the expected returns. If, because of policy changes, s_5 is rarely visited, the aberration at s_2 will disappear. Unfortunately, as soon as the policy is changed so that s_5 begins to be encountered more frequently, the aberration reappears, and so on. Thus, the system oscillates from policy to policy, unable to converge on a stable one.

4 Consistent Representation Methods

The last few years has seen the development of several RL algorithms that deal with active perception. The Lion Algorithm [52] learns to control visual attention in a primitive deictic sensory-motor system; the CS-QL algorithm [41] learns efficient, task-specific sensing trees; and the G-Algorithm [13] learns to extract task-relevant bits from a large input vector. [5] In this section, we review each of these algorithms in turn. We then present the *Consistent Representation Method*, a generalized approach to adaptive perception which unifies each of these algorithms [54]. The unified view summarizes the basic assumptions and limitations of these algorithms, and suggests new algorithms which extend or combine pieces of the basic architecture in novel ways.

4.1 The Lion Algorithm

The Lion Algorithm was perhaps the first reinforcement learning algorithm specifically designed to address an adaptive perception task [51]. It was used to learn a simple manipulation task in a modified blocks world. The distinguishing feature of this task is that the agent is equipped with a sensory-motor system that provides it with only partial access to the environment. To learn the task, the agent must learn to focus its visual attention on relevant objects and select appropriate motor commands. The details of the task are as follows.

4.1.1 The task

The learning task is organized into a sequence of trails. On each trial, the agent is presented with a pile of blocks. A pile consists of a random number of blocks (ranging from 1 to 50) arranged in arbitrary stacks. Blocks are distinguishable only by color; they may be red, green, or blue. Each pile contains a single green block. The agent's goal is simply to pick up the green block as quickly as possible.

[5] Note that these methods differ from supervised feature selection methods [28] that rely on the presentation of preclassified samples. The present algorithms operate without explicit supervision in the context of an embedded reinforcement learning task.

If the robot achieves the goal before the trial's time limit expires it receives a fixed positive reward, otherwise it receives no reward. The dynamics of the environment are such that a block can be grasped only when it is uncovered and the agent's hand is empty. Thus in some cases it is necessary to unstack blocks to reach the goal. In this task the effects of block manipulating actions are completely deterministic.

What differentiates this task from other blocks world problems (and other reinforcement learning tasks) is the agent's sensory-motor system. Instead of assuming a sensory system that provides a complete and objective description of every object in the scene, the system is equipped with a deictic sensory-motor system which provides the controller with an ability to flexibly access a limited amount of information about the scene at a time [1]. In a deictic sensory-motor system, selective perception is implemented using markers [45, 1]. Conceptually, a marker corresponds to a focus of attention. In practice, markers are used to establish reference frames for both perception and overt action. On the sensory side, placing a marker on an object in the environment, brings information about that object into view (i.e., into the internal representation). On the motor side, marker placement is used to select targets for overt manipulation. A specification for the sensory-motor system used by the agent is given in Figure 7. This system employs two markers, called the action-frame marker and the attention-frame marker. On the sensory side, the system generates a 20-bit input vector at each point in time. Most of these bits represent local, marker-specific information, such as the color and shape of a marker's bound object. Other bits detect relational properties such as vertical and horizontal alignment, while others detect spatially non-specific properties such as the presence or absence of red in the scene. By moving markers from object to object the agent can multiplex a wide range of information into its relatively small input bit register.

Listed on the right-hand side of Figure 7 are the internal motor commands supported by the sensory-motor system. These commands are partitioned into two groups, those related to the action-frame marker and those related to the attention frame marker. Both groups contain commands for controlling marker placement. These actions index objects by their primitive features (e.g., color) or by spatial relationship (e.g., top-of-stack). The action-frame marker has additional commands that are used for manipulating blocks. The "grasp-object-at-action-frame" command causes the system to grasp (if possible) the object marked by the action-frame marker. Similarly, the "place-object-at-action-frame" command causes the system to place a held object at the location marked by the action-frame.

The decision problem facing the agent's embedded controller is non-Markov since improper placement of the system's markers fails to multiplex relevant information onto the agent's internal representation. This point is illustrated in Figure 8 which shows two different external world states (each corresponding to a different states in a Markov model of the task) that, because of an improper placement of markers, generate the same internal representation.

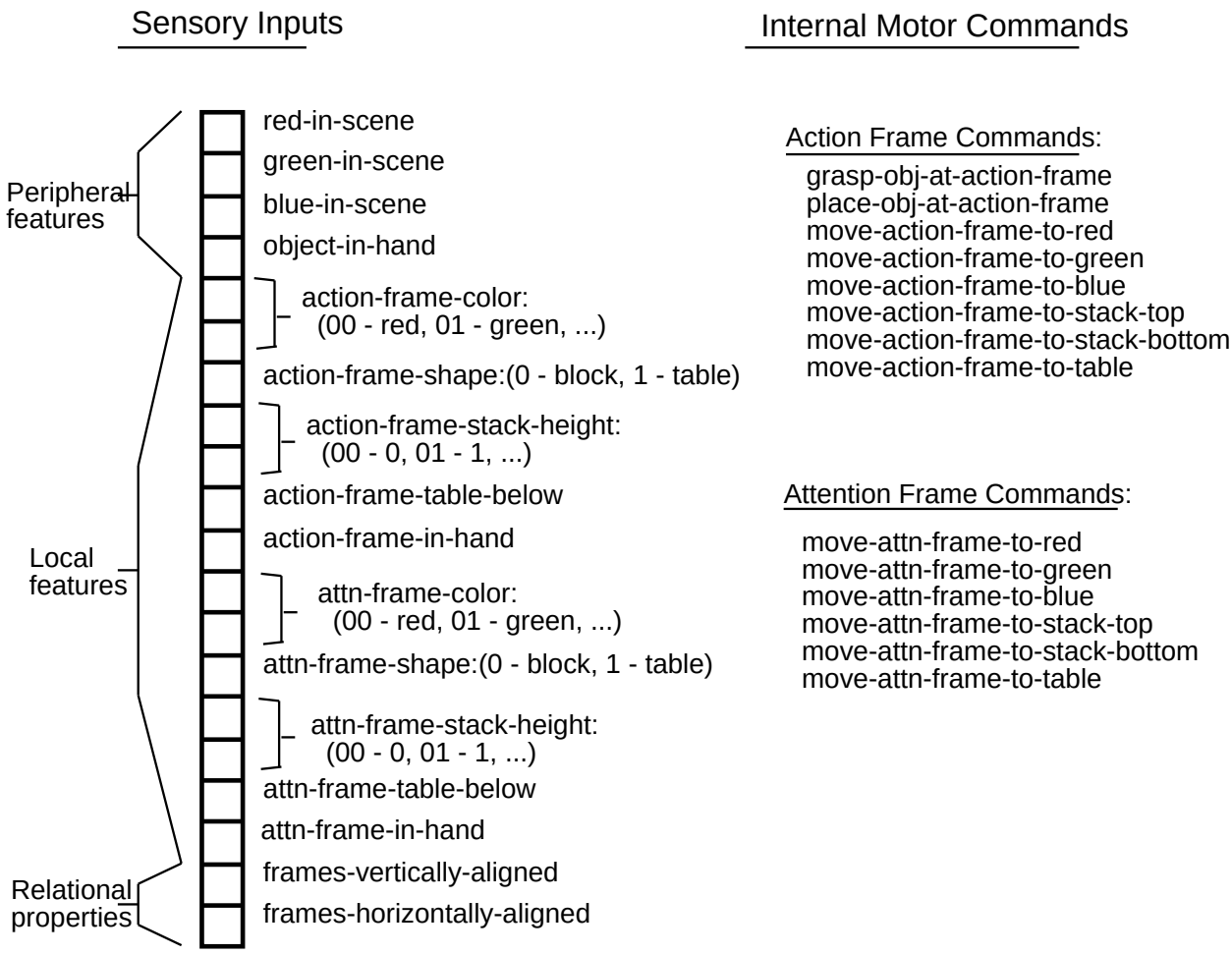


Figure 7: A specification for the deictic sensory-motor system used by Meliora-II in (Whitehead, 1991). The system has two markers, an *action-frame* marker and an *attention-frame* marker. The system has a 20-bit input vector, 8 overt actions, and 6 perceptual actions. The values registered in the input vector and the effects of internal action commands depend upon the bindings between markers in the sensory-motor system and objects in the environment.

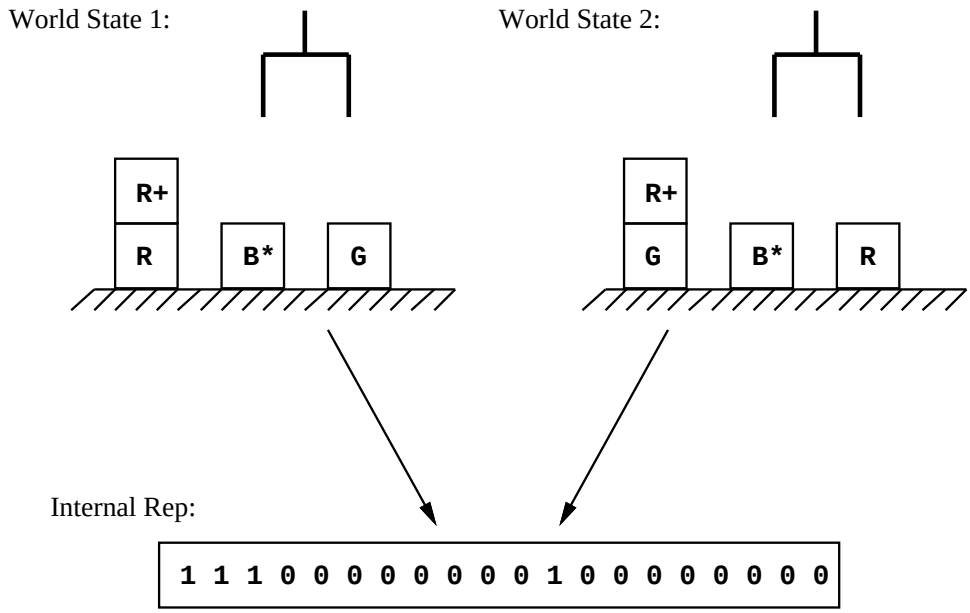


Figure 8: An Example of perceptual aliasing in the Block Stacking domain. In this case, two world states with different utilities and optimal actions generate the same internal representation.

4.1.2 Control

To tackle this non-Markov decision problem, the Lion algorithm adopts an approach which attempts to select overt (manipulative) actions based only on the action-values of internal states that are Markov. To accomplish this, the Lion algorithm breaks control into two stages. At the beginning of each control cycle a perceptual stage is performed. During the perceptual stage, a sequence of commands for moving the attention-frame marker are executed. These so-called "perceptual actions" cause a sequence of input vectors to appear in the input register. These values are temporarily buffered in a short term memory. Since perceptual actions do not change the state of the external environment, each buffered input corresponds to a representation of the current external state. If the perceptual actions are selected with care one of these internal states will be Markov (i.e., will encode all information relevant to selecting the optimal action). Once the perceptual stage is completed, the overt stage begins. During the overt stage an action for changing the state of the external environment is selected. These so-called "overt actions" correspond to commands for the action-frame marker.[6] To guide selection of an overt action, the Lion algorithm maintains a special action-value function which is defined over internal-state, overt-action pairs. This overt action-value function is

[6] Notice that moving the action-frame marker from one object to another changes the state of the external environment since it changes the set of objects that can be effected by the grasp and place commands.

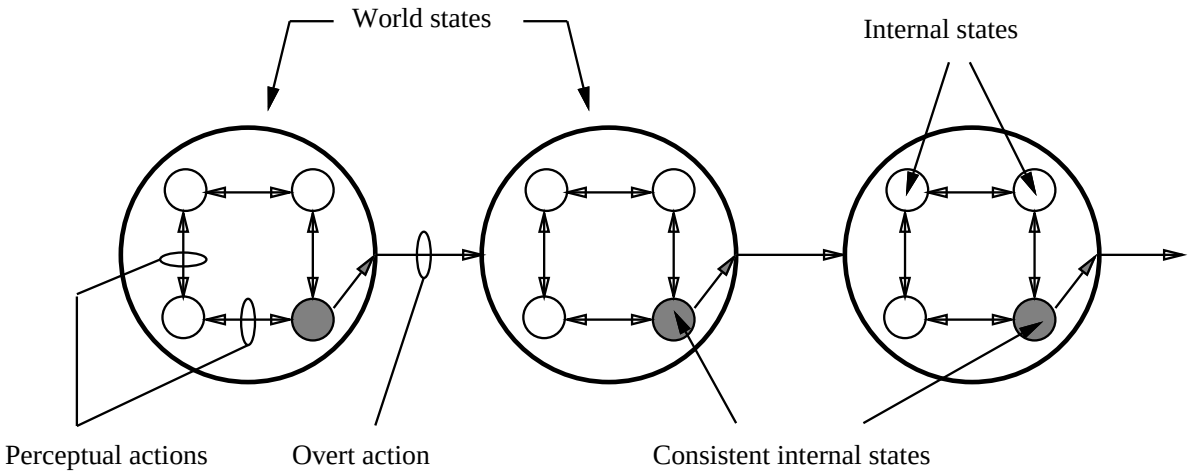


Figure 9: A graphical depiction of the Lion Algorithm. The large (super) graph depicts the overt control cycle, where large nodes correspond to world states and arcs correspond to overt actions. The subgraphs embedded within each large node depict perceptual cycles, with nodes corresponding to internal representations of the current world state and arcs corresponding to perceptual actions.

special in that the action-values for non-Markov states are suppressed (i.e., ideally they are equal to zero), whereas the action-values for Markov states are allowed to take on their nominal values. Given this action-value function, the Lion algorithm, during the overt stage, selects an overt action by simply examining the action-values of each buffered internal state and choosing the action that is maximal. Since non-Markov states tend to have suppressed action-values, the selected action tends to correspond to the maximal action from a Markov internal state. Figure 9 illustrates this two stage control cycle graphically.

4.1.3 Learning

A special learning rule is used to learn the overt action-value function. The learning procedure operates as follows. First, the internal state with the maximal action-value is identified as the *Lion*. The action-value for this state is updated according to the standard rule for 1-step Q-learning (viz. Equation 10). Next, the error term in the updating rule for the Lion state is used to update the action-values for the remaining buffered states. This is done so that once an accurate action-value is learned for a Markov state, further changes in the action-values for non-Markov states cease. Finally, each buffered state is tested to see if it is non-Markov. If a state tests positive, its action-value is reset to zero.

A very simple procedure is used to identify potentially non-Markov states. The rule simply examines the sign of the error term in the 1-step Q-learning rule (that is, the sign of the difference between a state's current action-value and the action-value estimate constructed after a one step delay). If all action-values are initially

zero, the task is deterministic, and all rewards are positive, then non-Markov states, due to perceptual aliasing, tend to regularly overestimate their action-values (i.e., show a negative error), whereas Markov states tend to monotonically approach the optimal action-value from below (i.e., positive error only). Therefore, non-Markov states can be detected by monitoring the sign in the estimation error.[7]

The learning rule for the perceptual stage is much simpler. For perceptual control a *perceptual action-value function* is estimated over internal-state, perceptual-action pairs. During the perceptual stage, perceptual actions are selected by choosing the action that maximizes the action-value for the current input bit vector (internal state). The perceptual action-value function is updated within the perceptual stage, using the standard 1-step Q-learning rule except that the overt utility of the internal state is also accounted for. Since non-Markov states tend to have suppressed overt action-values, perceptual actions that tend to lead to Markov internal states tend to have higher action-values than those that do not. (See [50] for further details).

4.1.4 Discussion

The Lion algorithm is able to learn the block manipulation task described above. It learns a perceptual control strategy that focuses the attention frame marker on the green block, and learns an overt control policy that moves the action-frame marker as needed to unstack covering blocks. Detailed experimental results can be found in [50]. Nevertheless the assumptions exploited by the Lion algorithm make it applicable only to tasks that meet the following restrictions:

1. The effects of actions must be deterministic;
2. Only positive rewards are allowed;
3. For each external state, there must exist at least one configuration of the sensory system that generates an internal state that is Markov.

4.2 CS-QL

Most work in machine learning aimed at learning classification tasks focuses only upon the predictive power of a given piece of information, and neglects to account for the cost of obtaining it [28, 3]. Tan recognized that to learn classification procedures that are efficient it is necessary for the learning algorithm to explicitly account for the cost of sensing. In [41] he develops two cost-sensitive learning algorithms for classification tasks: CS-ID3 and CS-IBL, respectively [41]. CS-QL, which stands for Cost-Sensitive Q-Learning, resulted when he combined ideas for cost-sensitive learning with reinforcement learning [40]. In CS-QL, the reinforcement learning

[7] Subtle interactions sometimes cause Markov states to overestimate their action-values. This sometimes leads to suppression of Markov states. However these states tend to bounce back from such suppressions and eventually stabilize. For a detailed discussion of this technique for detecting non-Markov states see [50] and [41].

agent not only learns the overt actions needed to perform a task, but also learns an efficient procedure for classifying the current state of the environment with respect to the task.

CS-QL and the Lion algorithm share the same basic control cycle. That is, in CS-QL control is decomposed into a two stage process of sensing (perceptual control) and action (overt control). However, the sensing model used in CS-QL is considerably different. Instead of using a deictic sensory-motor system, CS-QL adopts a sensing model in which the agent is equipped with a set of atomic sensing tests. Each sensing test provides a specific piece of information about the external environment.[8] Also, instead of learning a perceptual control policy, as in the lion algorithm, CS-QL constructs a classification tree, where internal nodes correspond to sensing operations, branches correspond to test results, and leaves correspond to the states in the agent's internal representation. In CS-QL, the agent has learned an adequate classification tree when every leaf in the tree is Markov; that is, when each leaf represents a unique state in a Markov model of the task.

The classification tree is learned incrementally. Initially, the tree consists of a single root node. As non-Markov leaves are detected, they are expanded (converted to internal nodes) by attaching sensing operations to them. The new leaf nodes that result introduce new distinctions into the representations. The tree is expanded until a Markov representation is achieved.

When expanding a node, CS-QL simply selects the least expensive sensing operation, among those that remain, to attach to the target leaf. This heuristic favoring low-cost tests tends to explore inexpensive sensing procedures first, but may not always generate the most efficient trees. By incorporating a more sophisticated selection method that accounts for both cost and the discriminatory power of each sensing test (See the G-algorithm below) more efficient classification trees should result. To detect non-Markov leafs, CS-QL uses the same overestimation principle employed by the Lion algorithm. Thus, CS-QL is also limited to deterministic domains.

CS-QL has been demonstrated successfully in a simulated robot navigation task similar to the one shown in Figure 3. However, unlike other navigation tasks studied in reinforcement learning (e.g., [39, 53, 19]), the robot is not automatically provided with knowledge of its current position. Instead, it must employ its sensors to gather information about its local surround and deduce its position from the surrounding geographic structure. The robot's sensing operations allow it to detect properties (e.g., empty, barrier, cup) of nearby cells in the maze. The cost of sensing a cell is assumed to be proportional to its distance from the robot. By accumulating features from nearby cells the system can successfully identify its position withing the maze. An example of a classification tree learned by CS-QL, along with several state descriptors are shown in Figure 10.

[8] Note, tests are atomic in that they cannot be composed in any meaningful way.

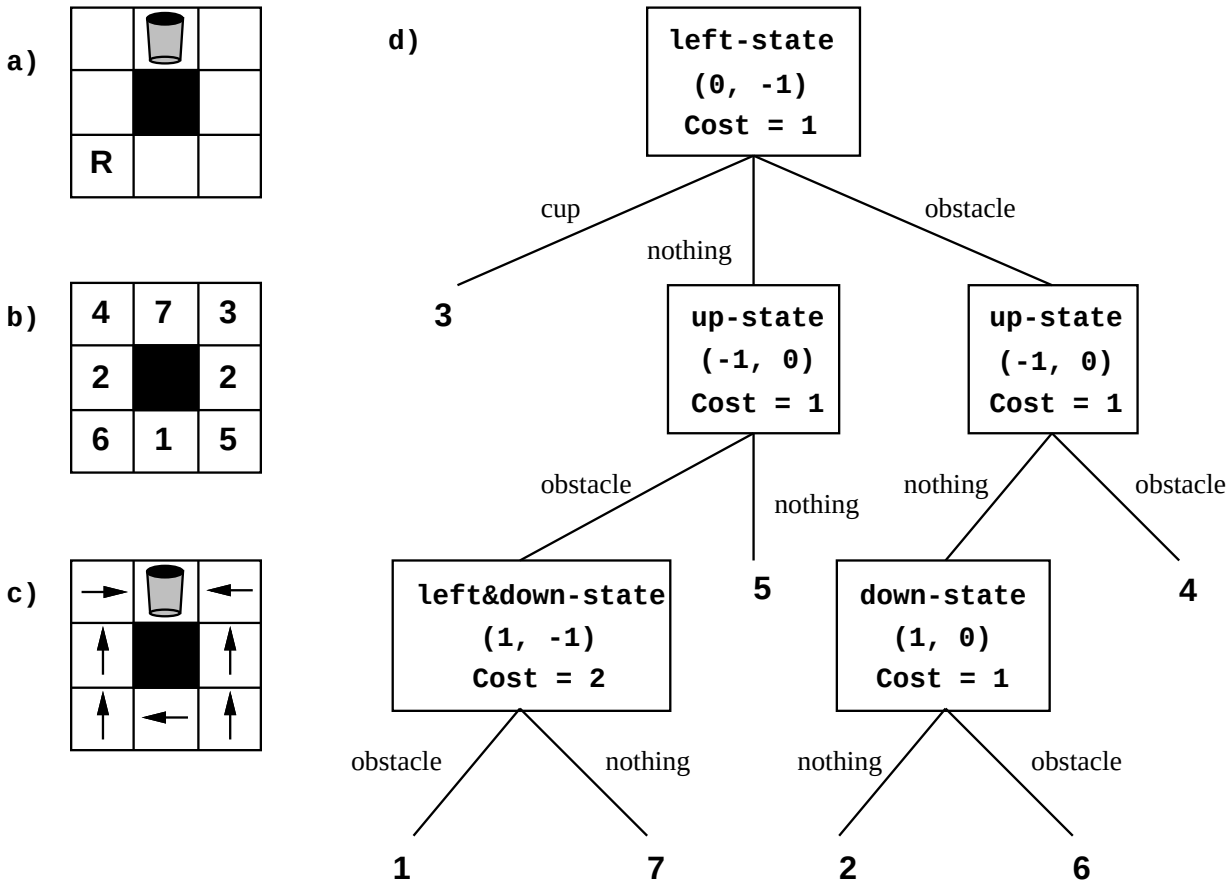


Figure 10: A simple example of CS-QL: (a) a 3 by 3 grid world, (b) a learned mapping between state descriptions and states, (c) a learned optimal decision policy, and (d) a learned cost-sensitive classification tree. (Reproduced with permission).

4.3 The G-Algorithm

The G-algorithm is a third technique developed to address a kind of adaptive perception task. However, unlike the Lion algorithm and CS-QL, its development was not specifically motivated by the desire to minimize the cost of sensing or by the need to control an active sensory system. Instead the G-algorithm was developed to mitigate problems caused by the availability of too much information. In particular, when Chapman and Kaelbling tried to apply Q-learning to learn a simple align-and-shoot subtask in the context of a more general video-game domain (called Amazon), they found the the learning system was being overwhelmed by the shear volume of information generated by the sensory system. The subtask involves aligning the agent with a target, orienting to it, and firing a weapon. At each point in time, the agent's sensory system generates 100 bits of input. Using all this information results in an internal state space containing 2^{100} states. Most of the bits in the input are irrelevant to *this* specific task and just interfere with learning by introducing unnecessary distinctions in the internal representation. On the other hand, the bits that are specifically relevant are not necessarily known ahead of time and at other stages of the game, the "irrelevant bits" are vitally important. The G-algorithm was developed to learn a control policy which could generalization over irrelevant information in input.

The G-algorithm works by identifying bits in the input vector that are significant/important to control. It is very similar to CS-QL in that both incrementally grow classification trees. That is, both start with an single root node (i.e., assuming no information is relevant), then construct a tree-structure classification circuit by recursively splitting nodes based on the values of sensory inputs. In CS-QL, the information used to split nodes in the tree corresponds to the results of sensing acts (or tests), in the G-algorithm nodes are split based on the values of bits in the input. As in CS-QL, the leaves of the G-algorithm's tree define the agent's internal state space. Unlike CS-QL, the G-algorithm does not associate a cost with sensing/reading a bit.

What sets the G-algorithm apart from both CS-QL and the Lion algorithm, is the method it uses to detect non-Markov internal states. CS-QL and the Lion algorithm both monitor the sign in estimation error to detect non-Markov states; a method that is limited to deterministic tasks only. The G-algorithm uses a much more general statistical test. In general, a leaf in the classification tree is non-Markov if it can be shown there are bits in the input vector (that have not already been tested in traversing the tree from node-to-leaf) that are statistically relevant to predicting future rewards. To detect if a leaf is non-Markov, the G-algorithm uses the Student's T-test [36] to find statistically significant bits. That is, over time as the agent experiences a variety of state, and for a given bit and a give leaf, situations that are classified into the leaf are divided into two blocks. One block corresponds to situations where the bit in question is on, the other when the bit is off. Data about the occurrence of reward (either immediate or future) is collected for each block. Given these two sets of data, a Student's T-test is used to determine how probable it is that distinct distributions gave rise to them. If after sufficient

sampling, this probability estimate is above a threshold, the bit is deemed relevant and the leaf is split.

The insight provided by the G-algorithm is to use statistical methods to test bit relevance (and consequently detect non-Markov states). The specific algorithm is limited in that the T-test assumes that the underlying distributions being compared are Gaussian. This is clearly not the case in general, since reward distributions can be arbitrary. However, this problem can be mitigated by comparing distributions of *cumulative* rewards which (via the central limit theorem) tend toward normality as the number summed increases. Also, the G-algorithm is not guaranteed to detect bits that are relevant in higher order pairings. A bit's relevance must be apparent in isolation. Finally, additional memory and sensing is required to gather statistics for relevance testing. Nevertheless these difficulties and limitations seem to be a minor price to pay for a method that extends to stochastic domains.

The G-algorithm was successfully demonstrated on the orient-and-shoot task. In particular, it was found to significantly outperform an alternative approach that used error backpropagation in a neural network. See [13] for details and a discussion of some difficulties they did encounter.

4.4 The Consistent Representation Method

While the algorithms described above vary considerably in their detail, they all share the same basic approach. We refer to this common framework as the *Consistent Representation (CR) Method*.[9] The key features of the CR-method are:

1. At each time step, control is partitioned into two stages: a perceptual stage followed by an action (or overt) stage.
2. The perceptual stage aims to generate an internal representation that is Markov.
3. The action stage generates (external) state modifying actions in an effort to maximize cumulative reward
4. Learning occurs in both control stages. For the action-stage, traditional reinforcement learning techniques are used. These techniques impose a Markov constraint on the internal state space. This constraint, in turn, drives adaptation in the perceptual stage in that the perceptual stage constantly monitors the internal representation for non-Markov states. When one is found, the perceptual process is modified to eliminate it.
5. It is assumed that the external state can always be identified from immediate sensory inputs.

[9] The term *Consistent Representation* is derived from the fact that it is not strictly necessary for the internal state space to be absolutely Markov. In particular, it is sufficient for each state to be Markov with respect to predicting future rewards (but not necessarily future states). This slightly weaker concept of being "partially Markov" or "Markov with respect to reward" has been associated with the term "consistent". See [50] for a further discussion of this distinction.

Figure 11 illustrates an architectural embodiment of the CR-method. The major components include: a selective sensory-motor interface, a perception module, a controller module, and a representation monitor. The line from the perception module to the sensory-motor interface represents perceptual control (or selection) acts. The line from the controller module to the sensory-motor interface represents overt acts. Both the controller and the perceptual modules are adaptive. Reward from the environment is received by both the controller and the representation monitor. The representation monitor detects non-Markov states and provides feedback to the perception module.

The correspondence between the components of this architecture and each of the previous algorithms is as follows. The Lion algorithm assumes a deictic sensory-motor system which includes commands for moving perceptual (or attentional) markers; CS-QL assumes a sensory-motor interface that consists of a set of discrete sensing acts; and the G-algorithm assumes a binary input vector from which individual bits are selected. The identification procedure implemented in the perception module takes the form of a "perceptual policy" in the Lion algorithm, and the form of a binary classification tree in CS-QL and the G-algorithm. The task-specific internal representation generated by the Lion algorithms corresponds to a subset of input bit vectors; while in CS-QL and the G-algorithm it is defined by the leaves of a classification tree. The Lion, CS-QL, and G-algorithm all use a form of Q-learning for overt control. For representation monitoring, both the Lion and CS-QL algorithm use an overestimation technique, while the G-algorithm relies on a more general statistical method.[10]

Relating the Lion, CS-QL and G-algorithms in the common framework of the CR-method is useful for two reasons. First, it promotes cross-fertilization of ideas between specific algorithms. For instance, the statistical methods used by the G-algorithms can be incorporated into Lion and CS-QL to yield algorithms that function in stochastic domains. Second, the structure provided by the CR-method highlights shared assumptions and limitations, and it suggests extensions to overcome them. In particular, a fundamental assumption made by all these algorithms is that all external states can be identified at each point in time from immediate sensor inputs. This assumption makes these techniques inappropriate for many interesting tasks that require memory to keep track of information that for one reason or another has become perceptually inaccessible. These more general hidden state tasks and several memory-based approaches to them are the subject of the next section.

[10] A version of the Lion algorithm has also been developed where feedback from an external supervisor is used to detect non-Markov states. This external supervision dramatically improves both perceptual and overt learning [50].

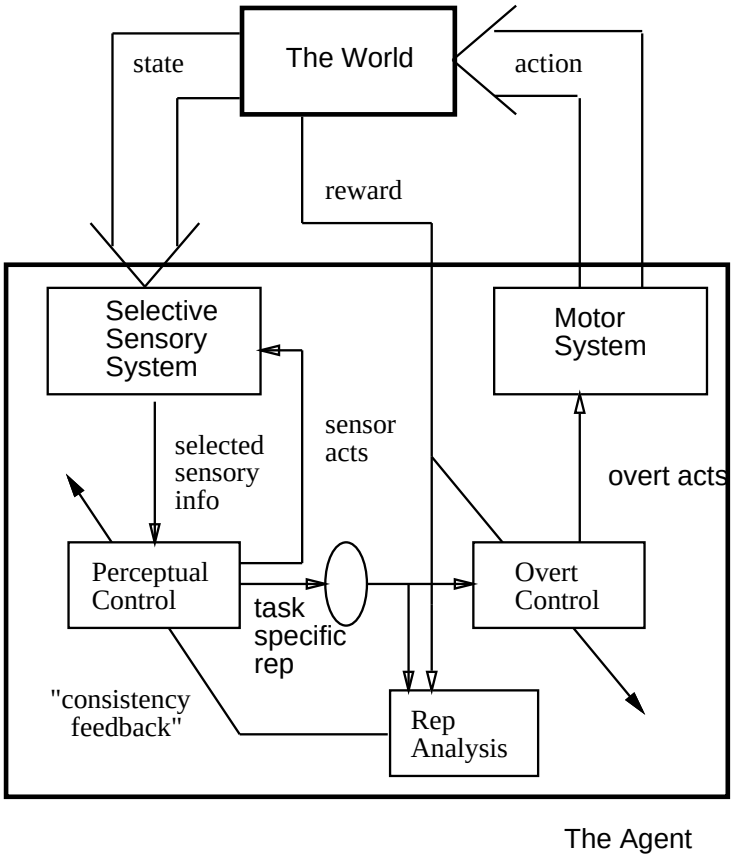


Figure 11: The basic architecture of a system using the Consistent Representation Method. Control is accomplished in two stages: a perceptual stage, followed by an overt stage. The goal of the perceptual stage is to generate a Markov, task-dependent internal state space. The goal of overt control is to maximize future discounted reward. Both control stages are adaptive. Standard reinforcement learning algorithms can be used for overt learning, while perceptual learning is driven by feedback generated by a representation analysis module, which monitors the internal state space for non-Markov states.

5 Memory-Based Methods

One obvious approach to dealing with inadequate perception and non-Markov decision problems is to to allow the agent to have a memory of its past. This memory can help the agent identify hidden states, since it can use differences in memory traces to distinguish situations that based on immediate perception appear identical. The problem is: given a huge volume of information available about the past, how should the agent decide what to remember, how to encode it, and how to use it. There are two approaches to this problem that have been discussed in the literature. In one approach the agent keeps a sliding window of its history, in the other approach the agent builds a state-dependent predictive model of environmental observables [33, 43, 5, 14, 23]. In addition to these two approaches, this section describes a new third approach, which learns a history-sensitive control policy directly from reinforcement.

5.1 Three Memory Architectures

Figure 12 depicts three memory architectures for reinforcement learning in non-Markov domains. In all three architectures a neural network (Q-net) is trained using temporal difference methods to incrementally learn an action-value function (Q-function).

In the *window-Q architecture*, instead of relying only upon immediate sensory inputs (or sensations) to define its internal representation, this architecture uses its immediate sensations, the sensations for the N most recent time steps, and the N most recent actions to represent its current state. In other words, the window-Q architecture allows direct access to the information in the past through a sliding window. N is called the *window size*. The window-Q architecture is simple and straightforward. However, to use this architecture one must choose a window size, which may be difficult to do in advance. On the one hand, if the selected window size is too small, the internal representation may not be sufficient to define a state space that is Markov. On the other hand, an em input generalization problem may arise if the window size is chosen to be too large, or if the window must necessarily be large to capture relevant information that is sparsely distributed in time. Under these circumstances excessive amounts of training may be required before the neural network can accurately learn the action-value function and generalize over the irrelevant inputs. In spite of these problems, the window-Q architecture is worthy of study, since 1) this kind of *time-delayed neural network* ahs been found to be useful in speech recognition tasks [46], and 2) the architecture can be used to establish a baseline for comparing other methods.

The window-Q architecture is sort of a brute force approach to using memory. An alternative is to distill a (small) set of *contextual features* out of the large volume of information about the past. This historical context together with the agent's current sensory inputs can then be used to define its internal representation. If the context features are constructed correctly then the resultant internal state space will be Markov and standard RL methods be used to learn an optimal control

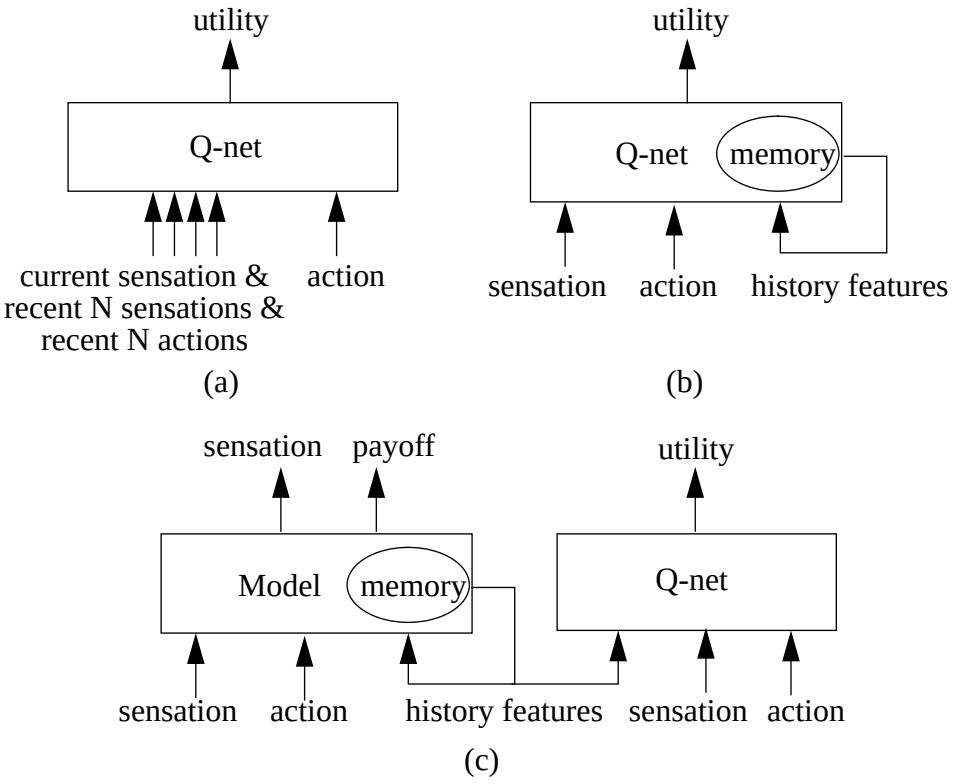


Figure 12: Three memory architectures for reinforcement learning in non-Markov domains: (a) window-Q architecture, (b) recurrent-Q architecture, and (c) recurrent-model architecture.

policy. The *recurrent-Q* and *recurrent-model* architectures illustrated in Figure 12 are based on this basic idea. However, they differ in the way they construct their context features. Unlike the window-Q architecture, both of these architectures can in principle discover and utilize historical information that depend on sensations arbitrarily deep in the past, although in practice this has been difficult to achieve.

Recurrent neural networks, such as Elman networks [16], provide one approach to constructing relevant context features. As illustrated in Figure 13, the input units of an Elman network are divided into two groups: the (immediate) sensory input units and the *context units*. The context units are used to encode a compressed representation of relevant information from the past. Since these units function as a kind of memory and encode an aggregate of previous network states, the output of the network depends upon past as well as current inputs.

The recurrent-Q architecture uses a recurrent network to estimate the action-value function directly. To predict action-values correctly, the recurrent network (called recurrent Q-net) must learn contextual features which enable the network

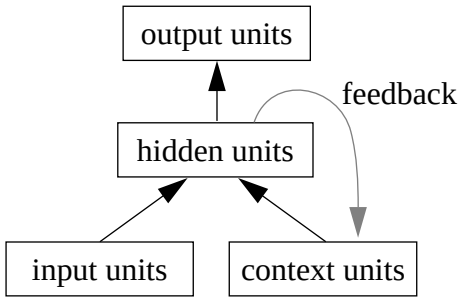


Figure 13: An Elman Network.

to distinguish between different external states that generate the same immediate sensory inputs.

The recurrent-model architecture (Figure 12c) consists of two concurrent learning components: a *1-step prediction module* or simply "the model," and a Q-learning module. The prediction module is responsible for learning to predict the immediate sensory inputs (and rewards) that result from performing an action. Because the agent's immediate inputs do not completely code the state of the external environment, the model must learn and use a context features to accurately predict the effects of an action on the the environment. If we assume that a accurate predictive model can be learned, and that the models context features can be extracted, then a Markov state space can be generated for the Q-learning component by defining its inputs (internal state space) to be the conjunction of the agent's immediate sensory input and the context features. This follows since, at any given time, the next state of the environment can be completely determined by this new state representation and the action taken.

In general, the predictive model must be trained to predict not only the new sensory inputs but also the immediate reward. To see why, consider a packing task which has 3 steps: put a gift into an open box, seal the box so that it cannot be opened again, place the box in the proper bucket depending on the color of gift in the box. Further, suppose a reward is given only when the box is placed in the correct bucket. Note that the agent is never required to know the gift color in order to predict future sensory inputs, since the box cannot be opened once sealed. Therefore a model that only predicts sensations may not have a set of context features adequate for control since it these features may not encode information about the color of the present.

Both the recurrent-Q and recurrent-model architectures learn context features using a gradient descent, least-mean-square method (e.g., error back-propagation), but they differ in an important way. In learning the predictive model, the goal is to minimize errors between actual and predicted sensory inputs and rewards. In this case, the environment provides all the needed training information, which is

consistent over time as long as the environment does not change. For recurrent Q-learning, the goal is to minimize errors between temporally successive predictions of action- and utility-values, (see Equation 10). In this case, the error signals are computed based partly on information from the environment and partly on the agent's current estimate of the optimal action-value function. Since this latter term changes over time and carries little or no useful information during the early stages of learning, these error signals may be in general weak, noisy, and even inconsistent over time. Because of this the practical viability of the recurrent-Q architecture is uncertain.

Having introduced these architectures, it is worthwhile to note that combinations of these approaches are also possible. For example, we can combine the first two architectures: the inputs to the recurrent Q-net could include not just the current sensory input but also recent inputs and recent actions. We can also combine the last two architectures. For instance one approach would be to share the context units between the model network and the Q-network such that the context features learned would based on prediction errors from both networks. Although there are many possibilities, this article is only concerned with the three basic architectures. Further investigation is needed to see if other combinations will result in better performance than the basic versions.

5.2 Network Training

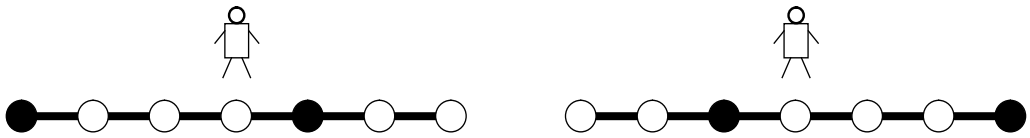
The (non-recurrent) Q-nets of the window-Q and recurrent-model architectures can be trained using a straightforward combination of temporal difference methods [38] and the connectionist back-propagation algorithm [31]. This combination has been successfully applied to solve several nontrivial reinforcement learning problems [20, 23, 42].

Training the model of the recurrent-model architecture is slightly more complicated. Recurrent networks can be trained by a recurrent version of the back-propagation algorithm called *back-propagation through time* (BPTT) or *unfolding of time* [31]. BPTT is based on the observation that any recurrent network spanning T steps can be converted into an equivalent feed-forward network by duplicating the network T times. Once a recurrent network is unfolded, back-propagation can be directly applied. The Q-net of the recurrent-Q architecture can also be trained by BPTT together with temporal difference. For detailed network structures and implementation, see [23].

5.3 Simulation Results

This subsection presents experimental results of a study in which the three memory-based architectures were applied to a series of non-Markov decision tasks. Through this study, we have gained insight into the behavior of these architectures, and a better understanding of the relative merits of each and the conditions for their useful application. (Detailed descriptions of the simulation and results can be found in [23].)

2 possible initial states:



3 actions: walk left, walk right & pick up
4 binary inputs: left cup, right cup, left collision & right collision
reward: 1 when the last cup is picked up
0 otherwise

Figure 14: Task 1: A 2-cup collection task.

5.3.1 Task 1: 2-Cup Collection

We begin with a simple 2-cup collection task (Figure 14). This task requires the learning agent to pick up two cups located in a 1-D space. The agent has 3 actions: walking right one cell, walking left one cell, and pick-up. When the agent executes the pick-up action, it will pick up a cup if and only if the cup is located at the agent's current cell. The agent's sensory input includes 4 binary bits: 2 bits indicating if there is a cup in the immediate left or right cell, and 2 bits indicating if the previous action results in a collision from the left or the right. An action attempting to move the agent out of the space will cause a collision.

The cups are placed far enough apart that once the agent picks up the first cup, it cannot see the other one. To act optimally, the agent has to somehow remember the location of the second cup. This task is non-trivial for several reasons: 1) the agent cannot sense a cup in front of it, 2) the agent gets no reward until both cups are picked up, and 3) the agent often operates with no cup in sight especially after picking up the first cup. In this experiment, each trial begins in one of two possible initial states, as shown in Figure 14. This restriction simplifies the task by avoiding perceptual aliasing at the onset of a trial when no history information is available.

The three memory architectures were tested on this cup collection task. The experiment was repeated 5 times, and every time each successfully learned an optimal control policy within 500 trials. (The window size N was 5.) One interesting observation, however, was the following: The recurrent-model architecture never learned a perfect model within 500 trials. For instance, if the agent has not seen a cup for 10 steps or more, the model normally is not able to predict the appearance of the cup. But this imperfect model did not prevent Q-learning from learning an optimal policy.

This experiment revealed two lessons:

- All of the three architectures worked for this simple cup-collection problem.
- For the recurrent-model architecture, just a partially correct model may provide sufficient context features for optimal control. This is good news, since a perfect model is often difficult to obtain.

5.3.2 Task 2: Task 1 With Random Features

Task 2 is simply Task 1 with two random bits in the agent's sensation. The random bits simulate two difficult-to-predict and irrelevant features accessible to the learning agent. In the real world, there are often many features which are difficult to predict but fortunately not relevant to the task to be solved. For example, predicting whether it is going to rain outside might be difficult, but it does not matter if the task is to pick up cups inside. The ability to handle difficult-to-predict but irrelevant features is important for a learning system to be practical.

The simulation results are summarized as follows: The two random features gave little impact on the performance of the window-Q architecture or the recurrent-Q architecture, while the noticeable negative impact on the recurrent-model architecture was observed.

The system using the recurrent-model architecture exhibited streaks of optimal performance during the course of 300 trials. However, it apparently could not stabilize on the optimal policy; it oscillated between the optimal policy and several sub-optimal policies. It was also observed that the model tried in vain to reduce the prediction errors on the two random bits. There are two possible explanations for the poorer performance compared with that obtained when there are no random sensation bits. First, the model might fail to learn the context features needed to solve the task, because much of the effort was wasted on trying to learn to predict the random bits. Second, because the activations of the context units were shared between the model network and the Q-net, a change to the representation of context features on the model part could simply destabilize a well-trained Q-net, if the change was significant. The first explanation is ruled out, since the optimal policy indeed was found many times. To test the second explanation, we fixed the model at some point of learning and allowed only changes to the Q-net. In such a setup, the agent found the optimal policy and indeed stuck to it.

This experiment revealed two lessons:

- The recurrent-Q architecture is more economic than the recurrent-model architecture in the sense that the former will not try to learn a context feature if it does not appear to be relevant to predicting action-values.
- A potential problem with the recurrent-model architecture is that changes to the representation of context features on the model part may cause instability on the Q-net part.

5.3.3 Task 3: Task 1 With Control Errors

Noise and uncertainty prevail in the real world. To study the capability of these architectures to handle noise, we added 15% control errors to the agent's actuators,

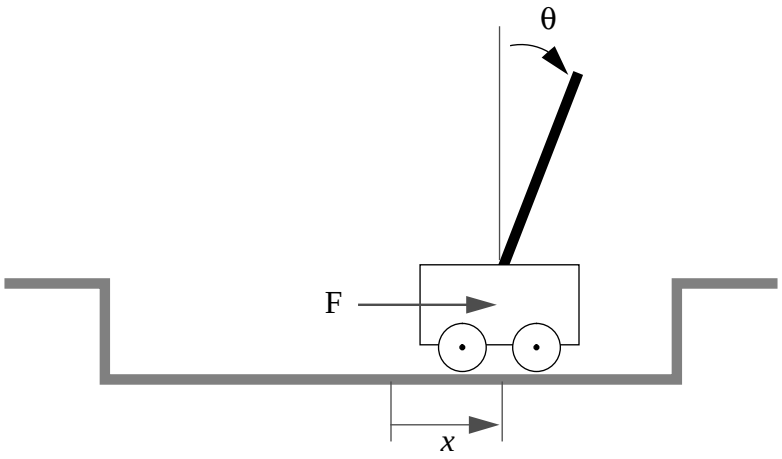


Figure 15: The pole balancing problem.

so that 15% of the time the executed action would not have any effect on the environment. (The 2 random bits were removed.)

In 3 out of the 5 runs, the window-Q architecture successfully found the optimal policy, while in the other two runs, it only found suboptimal policies. In contrast, the recurrent-Q architecture always learned the optimal policy (with little instability).

The recurrent-model architecture always found the optimal policy after 500 trials, but again its policy oscillated between the optimal one and some sub-optimal ones due to the changing representation of context features, much as happened in Task 2. If we can find some way to stablize the model (for example, by gradually decreasing the learning rate to 0 at the end), we should be able to obtain a stable and optimal policy.

Two lessons have been learned from this experiment:

- All of the three architectures can handle small control errors to some degree.
- Among the architectures, recurrent-Q seems to scale best in the presence of control errors.

5.3.4 Task 4: Pole Balancing

In the pole balancing problem, the system's objective is apply forces to the base of a movable cart in order to balance a pole that is attached to the cart via a hinge (Figure 15). This problem has been studied widely in the reinforcement learning literature. It is of practical interest because of its resemblance to problems in aerospace (e.g., missile guidance) and robotics (e.g., biped balance and locomotion). It is of theoretical interest because of the difficult credit assignment problem which

arises due to sparse reinforcement signals. In particular, in most formulations of the problem, the system only receives non-zero reinforcement when the pole falls over. For instance in our simulations the system receives a penalty of -1 when the pole tilt exceeds 12 degrees from vertical.

In the traditional pole balancing task, the system's sensory inputs include the position and velocity of the cart and the angular position and velocity of the pole [37]. This information completely characterizes the state of the system and yields control problem that is Markov. In our experiments, only the cart position and pole angle are given. This yields a non-Markov decision problem, and in order to learn an adequate control policy the system must construct contextual features resembling velocities for the cart and pole. In this experiment, a policy was considered satisfactory whenever the pole could be balanced for over 5000 steps in each of the 7 test trials where the pole starts with an angle of 0, ± 1, ± 2, or ± 3 degrees. (The maximum initial pole angle with which the pole can be balanced indefinitely is about 3.3 degrees.) In the training phase, pole angles and cart positions were generated randomly. The initial cart velocity and pole velocity are always set to 0. $N = 1$ was used here.

The input representation used here was straightforward: one real-valued input unit for each of the pole angle and cart position. The following table shows the number of trials taken by each architecture before a satisfactory policy was learned. These numbers are the average of the results from the best 5 out of 6 runs. (A satisfactory policy was not always found within 1000 trials.).

method	window-Q	recurrent-Q	recurrent-model
# of trials	206	552	247

A lesson has been learned from this experiment:

- While the recurrent-Q architecture was the most suitable architecture for the cup collection tasks, it was outperformed by the other two architectures for the pole balancing task.

5.4 Discussion

The above experiments provide some insight into the performance of the three memory architectures. This section considers task characteristics that may be useful in determining when one architecture may be preferred over another. Some of the features (or parameters) of a task that effect the applicability of these architectures are:

- **Memory depth.** One important problem parameter is the length of time over which the agent must remember previous inputs in order generate an internal representation that is Markov. For example, the memory depth for Task 1 is 2, as evidenced by the fact that the window-Q agent was able to obtain the optimal control based only on a window of size 2. The memory depth for the pole balancing task is 1. Note that learning an optimal policy may require a larger memory depth than that needed to represent the policy.

- **Payoff delay.** In cases where the payoff is zero except for the goal state, we define the payoff delay of a problem to be the length of the optimal action sequence leading to the goal. This parameter is important because it influences the overall difficulty of Q-learning. As the payoff delay increases, learning an accurate Q-function becomes increasingly difficult due to the increasing difficulty of credit assignment.
- **Number of context features to be learned.** In general, the more perceptual aliasing an agent faces, the more context features the agent has to discover, and the more difficult the task becomes. In general, predicting sensations (i.e., a model) requires more context features than predicting action-values (i.e., a Q-net), which in turn requires more context features than representing optimal policies. Consider Task 1 for example. Only two binary context features are required to determine the optimal actions: "*is there a cup in front*?" and "*is the second cup on the right-hand side or left-hand side*?". But a perfect Q-function requires more features such as "*how many cups have been picked up so far*?" and "*how far is the second cup from here*?". A perfect model for this task requires the same features as the perfect Q-function. But a perfect model for Task 2 requires even more features such as "*what is the current state of the random number generator*?", while a perfect Q-function for Task 2 requires no extra features.

  It is important to note that we do not need a perfect Q-function or a perfect model in order to obtain an optimal policy. A Q-function just needs to assign a value to each action in response to a given situation such that their relative values are in the correct order, and a model just needs to provide sufficient features for constructing a good Q-function.

5.4.1 Architecture Characteristics

Given the above problem parameters, we would like to understand which of the three architectures is best suited to particular types of problems. Here we consider the key advantages and disadvantages of each architecture, along with the problem parameters which influence the importance of these characteristics.

- **Recurrent-model architecture.** The key difference between this architecture and the recurrent-Q architecture is that its learning of context features is driven by learning an action model rather than the Q-function. One strength of this approach is that the agent can obtain better training data for the action model than it can for the Q-function, making this learning more reliable and efficient. In particular, training examples of the action model (<sensation, action, next-sensation, payoff> quadruples) are directly observable with each step the agent takes in its environment. In contrast, training examples of the Q-function (<sensation, action, utility> triples) are not directly observable since the agent must estimate the training utility values based on its own changing approximation to the true action-value function.

The second strength of this approach is that the learned features are dependent on the environment and independent of the reward function (even though the action model may be trained to predict rewards as well as sensations). As a result, these features can be reused if the agent has several different reward functions, or goals, to learn to achieve.

- **Recurrent-Q architecture.** While this architecture suffers the relative disadvantage that it must learn from indirectly observable training examples, it has the offsetting advantage that it need only learn those context features that are *relevant* to the control problem. The context features needed to represent the optimal action model are a superset of those needed to represent the optimal Q-function. This is easily seen by noticing that the optimal control action can in principle be computed from the action model (by using look ahead search). Thus, in cases where only a few features are necessary for predicting utilities but many are needed to predict completely the next state, the number of context features that must be learned by the recurrent-Q architecture can be much smaller than the number needed by the recurrent-model architecture.
- **Window-Q architecture.** The primary advantage of this architecture is that it does not have to learn the state representation recursively (as do the other two recurrent network architectures). Recurrent networks typically take much longer to train than non-recurrent networks. This advantage is offset by the disadvantage that the history information it can use are limited to those features directly observable in its fixed window which captures only a bounded history. In contrast, the two recurrent network approaches can in principle represent context features that depend on sensations that are arbitrarily deep in the agent's history.

Given these competing advantages for the three architectures, one would imagine that each will be the preferred architecture for different types of problems:

- One would expect the advantage of the window-Q architecture to be greatest in tasks where the memory depths are the smallest (for example, the pole balancing task).
- One would expect the recurrent-model architecture's advantage of directly available training examples to be most important in tasks for which the payoff delay is the longest (for example, the pole balancing task). It is in these situations that the indirect estimation of training Q-values is most problematic for the recurrent-Q architecture.
- One would expect the advantage of the recurrent-Q architecture — that it need only learn those features relevant to control — to be most pronounced in tasks where the ratio between relevant and irrelevant context features is the lowest (for example, the cup collection task with two random features). Although the recurrent-model architecture can acquire the optimal policy as long as just the relevant features are learned, the drive to learning the irrelevant features may cause problems. First of all, representing the irrelevant

features may use up many of the limited context units at the sacrifice of learning good relevant features. Secondly, as we have seen in the experiments, the recurrent-model architecture is also subject to instability due to changing representation of the context features— a change which improves the model is also likely to deteriorate the Q-function, which then needs to be re-learned.

The tapped delay line scheme, which the window-Q architecture uses, has been widely applied to speech recognition [46] and turned out to be quite a useful technique. However, we do not expect it to work as well for control tasks as it does for speech recognition, because of an important difference between these tasks. A major task of speech recognition is to find the temporal structure that already exists in a given sequence of speech phonemes. Whereas in reinforcement learning, the agent must look for the temporal structure generated by its own actions. If the actions are generated randomly as it is often the case during early learning, it is unlikely to find sensible temporal structures within the action sequence so as to improve its action selection policy.

6 Discussion

In principle, the memory-based architectures described in the previous section are applicable to non-Markov tasks in general. This raises the question of whether or not they might be usefully applied to adaptive perception tasks. For example, can these memory-based architectures learn to control the a deictic sensory-motor system similar to the one described in Figure 7? As of this writing, the question remains open. In principle, the memory-based architectures should work. However, only experimental studies will tell for sure, and preliminary results cast a shadow of doubt. In particular, in [13], a neural network using backpropagation was tested on the adaptive perception task (described in Section 4.3) in which the agent, to learn efficiently, had to select the relevant features in a 100 bit input. The G-algorithm (an instance of a CR-method) was able to learn the task, but the backpropagation network could not. Apparently the neural network had difficulty dealing with the *noise* introduced by the irrelevant bits in the input. Similar, kinds of difficulties can be expected to arise in controlling an active sensory system where the system has access to a tremendous volume of irrelevant information. However, to be fair it should be noted that network experiments were preliminary, and used a simple version of the backpropagation algorithm. More sophisticated methods, such as those using momentum terms in the updating rule, may yield better results.

It may also be possible to extend the CR-method to deal with more general hidden state problems. One simple approach along these lines would be to extend an agent's selective sensory system to include remembered sensory-motor events. That is, instead of selecting bits of information from the current sensory input only, the system could also select bits from a memory trace of previous inputs and actions. This approach is similar to the Window-Q architecture in that a memory trace is maintained, however it differs in that only a relatively small amount of information would be selected at each point in time. Moreover, under this scheme it might be

possible to devise use (or reference-based) rules for updating the memory-trace in a way that would preserve relevant memories while dropping irrelevant ones.

Other architectures that combine features from both the CR-method and memory-based architectures may also be very useful. For example, one problem with the CR-method as it currently stands is that the system uses no information about the previous state of the environment when trying to identify the current state. In a sense the system re-identifies the state of the environment starting from "scratch" after each action. Knowledge of the last state and the most recent action could considerably reduce the effort required to identify the current state, since in most environments transitions between states tend to be local and predictable. Thus instead of "rediscovering" the state after each action, the agent could merely verify the current state, or in the worst case, identify the outcome from a limited number of possibilities.

In addition to further exploring variations on the above architectures, future work must also assess the scalability to these algorithms. These algorithms were derived from a desire to extend reinforcement learning beyond Markov decision problems and to problems that involve active perception and/or hidden state. At to some extent we have been successful. Nevertheless, the tasks we have explored remain painfully simple compared to the scale of problems required for truly autonomous, intelligent behavior. A few of the issues that must be addressed to achieve scalability include:

- *Learning Bias:* Reinforcement learning can be viewed as a kind of search through the space of possible control policies. If that search can be biased in an appropriate direction, learning can proceed much more quickly than it might otherwise. One approach to introducing bias into a learning agent is to allow it to interact with other intelligent agents performing similar tasks. Other agents can serve as role models, advice givers, instructors, critics, and supervisors, and in general can strongly bias an agent's learning. Simple versions of these methods have been demonstrated in the context of reinforcement learning and have produced significant improvements in learning time [53, 15, 22]. However, much more work is needed.

- *Intelligent Credit Assignment:* Credit assignment is the fundamental problem in reinforcement learning: *viz* given that the agent has received a payoff, which parts of the agent were responsible for generating that payoff and how should the system be changed to improve performance. Most RL algorithms solve this problem by making incremental changes to the system over the course of many, many trials. However, this can take a long time. If additional knowledge about the causal structure of the environment can be made available, more efficient credit assignment methods can be developed (e.g., see [59] for an example of this idea).

- *Increased State Space and Action Space Complexity:* To date much of the work in reinforcement learning has been on problems that are small compared to the control problems facing real robotic systems. For example, a walking robot may require precise (continuous) information from dozens of sensors, and

may need to control dozens for effectors. The combinatorics associated with such problems quickly overwhelm the simplest RL methods. Moreover, real problems (such as robot walking) often suffer from a kind of severe temporal credit assignment problem, in which control must be administered at a very fine grain, whereas feedback for learning arrives at a relatively course grain. This delay, combined with the increased scale of multi-dimensional tasks, leads to tasks that are impractical for most existing techniques (though see [27] for promising work in this direction).

- *Multi-purpose Behavior:* Another source of complexity arises when we consider agents that must coordinate their behavior in order to achieve multiple goals. Under these circumstances, an agent's internal state space may increase exponentially in the number of possible goals, and it is necessary to develop methods for managing this explosion carefully. (See [35, 54] for work in this direction).

Of course there are many other issues that stand between current technology and the development of intelligent autonomous agents, and reinforcement learning is no panacea. However, the autonomy afforded by reinforcement learning methods makes them likely to play an important role. Moreover, the ubiquity of perceptual aliasing and non-Markov decision tasks in autonomous control makes these issues central.

7 Conclusions

Intelligent control systems must deal with information limitations imposed by their sensors. When inadequate information is available from the agent's sensors or when the agent must actively control its sensors in order to select relevant features, the internal decision problem it faces is necessarily non-Markov. Learning these control tasks can be very difficult since traditional reinforcement learning methods typically yield poor performance.

In this article we have presented several approaches to dealing with non-Markov decision problems. The Consistent Representation Method was proposed as an approach to dealing with tasks that involve control/selection in an active sensory system. In the CR-method, control is partitioned into two phases: a perceptual control phase, which aim to identify the current state of the environment; and an overt control phase, which aims to control the state of the environment. Three instances of this method, the Lion algorithm [52], the G-algorithm [13], and CS-QL [40], were described and examples of their uses presented. The major assumption made by the CR-method is that the state of the environment can be identified at each point in time by appropriately controlling/selecting aspects of sensory system. This assumption prevents it from being applied to tasks in which relevant state information is temporarily hidden from view.

For these tasks memory-based methods are more appropriate. Three different memory-based architectures were described: window-Q, recurrent-model, and recurrent-Q. The window-Q architecture uses a tapped-delay line to maintain a

fixed length history of recent sensory-motor events. The recurrent-model architecture constructs predictive model of the external environment, whose own internal state is used, in conjunction with sensory inputs to drive control. The recurrent-Q architecture uses a recurrent neural network to learn the action-value function for the non-Markov task directly. Because the recurrent network can encode state information across time steps, its own internal state is used to resolve ambiguities caused by inadequate sensory input. These three architectures were demonstrated on a series of hidden-state tasks, and conditions for their useful application were discussed.

The methods described in this article are preliminary in that they have only been demonstrated on relatively simple tasks and they have not been extensively tested or compared in very complicated domains. Nevertheless, these algorithms represent a significant advance over traditional reinforcement learning algorithms, which do not address non-Markov tasks at all. Perhaps these rather modest algorithms will serve as stepping stones to more sophisticated and capable methods for dealing with the ubiquitous problems of hidden state.

References

[1] Philip E. Agre. *The Dynamic Structure of Everyday Life.* PhD thesis, MIT Artificial Intelligence Lab., 1988. (Tech Report No. 1085).

[2] Philip E. Agre. *The Dynamic Structure of Everyday Life.* Cambridge University Press, Cambridge, forthcoming.

[3] David Aha. Incremental, instance-based learning of independent and graded concept descriptions. In *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY, 1989. Morgan Kaufmann.

[4] John Aloimonons, Isaac Weiss, and Amit Bandyopadhyay. Active vision. *International Journal of Computer Vision*, 1(4):333–356, 1988.

[5] J.R. Bachrach. *Connectionist Modeling and Control of Finite State Environments.* PhD thesis, University of Massachusetts, Department of Computer and Information Sciences, 1992.

[6] R. Bajcsy and P. Allen. Sensing strategies. In *U.S.-France Robotics Workshop*, Univ. of Pennsylvania, Philadelphia, PA, November 1984.

[7] Dana H. Ballard. Animate vision. Technical Report 329, Department of Computer Science, University of Rochester, 1990.

[8] A.B. Barto, S.J. Bradtke, and S.P. Singh. Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57, University of Massachusetts, Amherst, MA, 1991.

[9] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuron-like elements that can solve difficult learning control problems. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

[10] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[11] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, 1987.

[12] Lashon B. Booker. Triggered rule discovery in classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, June 1989.

[13] David Chapman and Leslie Pack Kaelbling. Learning from delayed reinforcement in a complex domain. In *Proceedings of IJCAI*, 1991. (Also Teleos Technical Report TR-90-11, 1990).

[14] L Chrisman. Reinforcement learning with perceptual aliasing: The predictive distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 183–188. AAAI Press/The MIT Press, 1992.

[15] Jeffery Clouse and Paul Utgoff. A teaching method for reinforcement learning. In *Proceedings of the Ninth International Conference on Machine Learning*. Morgan Kaufmann, 1992.

[16] J.L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[17] John H. Holland. Escaping brittleness: the possibilities of general-purpose learning algortihms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach. Volume II*. Morgan Kaufmann, San Mateo, CA, 1986.

[18] Leslie P. Kaelbling. *Learning in Embedded Systems*. PhD thesis, Stanford University, 1990.

[19] Long-Ji Lin. Self-improving reactive agents: case studies of reinforcement learning frameworks. In *Proceedings of the First International Conference on the Simulation of Adaptive Behavior*, September 1990.

[20] Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 781–786. AAAI Press/The MIT Press, 1991.

[21] Long Ji Lin. Memory approaches to reinforcement learning in non-markov domains. Technical Report 138, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992.

[22] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[23] Long-Ji Lin and T.M. Mitchell. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, School of Computer Science, Carnegie Mellon University, 1992.

[24] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. Research Report RC 16359, IBM T.J. Watson Research Center, December 1990.

[25] D. Michie and R. Chambers. Boxes: An experiment in adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence 2*, pages 137–152. Oliver and Boyd, Edinburgh, 1968.

[26] Marvin L. Minsky. *Theory of Neural-Analog Reinforcement Systems and Its Application to The Brain-Model Problem.* PhD thesis, Princeton University, 1954.

[27] Andrew Moore. Variable resolution dynamic programming: efficiently learning action maps in multivariate real-values state spaces. In *Proceedings of the eighth international conference on machine learning*, pages 333–337, 1991.

[28] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[29] Rick L. Riolo. *Empirical Studies of Default Heirarchies and Sequences of Rules in Learning Classifier Systems.* PhD thesis, Dept. of Computer Science and Engineering, University of Michigan, 1988.

[30] S. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, New York, NY, 1983.

[31] David. E. Rumelhart, Geoffery. E. Hinton, and Ronald. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition; Vol. 2: Psychological and Biological Models (J. L. Mcclelland and D. E. Rumelhart Eds.)*, pages 318–362. MIT Press, Camgridge, MA, 1986.

[32] A. L. Samuel. Some studies in machine learning using the game of checkers. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 71–105. Krieger, Malabar, FL, 1963.

[33] J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 3*, pages 500–506. Morgan Kaufmann, 1991.

[34] Jurgen Schmidhuber. Making the world differentiable: on using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report Report FKI-126-90 (revised), Technische Universitat Munchen, 1990.

[35] Satinder Singh. Transfer of learning across compositions of sequential tasks. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 348–352. Morgan Kaufmann, 1991.

[36] George W. Snedecor. *Statistical Methods.* Iowa State University Press, Ames, Iowa, 1989.

[37] Richard S. Sutton. *Temporal Credit Assignment In Reinforcement Learning.* PhD thesis, University of Massachusetts at Amherst, 1984. (Also COINS Tech Report 84-02).

[38] Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[39] Richard S. Sutton. Integrating architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, Austin, TX, 1990. Morgan Kaufmann.

[40] Ming Tan. Cost sensitive reinforcement learning for adaptive classification and control. In *Proceedings of the Ninth International Conference on Artificial Intelligence*, 1991.

[41] Ming Tan. *Cost Sensitive Robot Learning.* PhD thesis, Carnegie Mellon University, 1991.

[42] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[43] S. Thrun and K. Moller. Planning with an adaptive world model. In D. S. Tourtezky and R. Lippmann, editors, *Advances in Neural Information Processing Systems 3.* Morgan Kaufmann, 1991.

[44] Sebastian Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University, 1992.

[45] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984. (Also in: Visual Cognition, S. Pinker ed., 1985).

[46] A. Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1:39–46, 1989.

[47] Chris Watkins. *Learning from delayed rewards.* PhD thesis, Cambridge University, 1989.

[48] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 1992.

[49] Steven D. Whitehead. Scaling in reinforcement learning. Technical Report TR 304, Computer Science Dept., University of Rochester, 1989.

[50] Steven D. Whitehead. *Reinforcement Learning for the Adaptive Control of Perception and Action.* PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, November 1991.

[51] Steven D. Whitehead and Dana H. Ballard. Active perception and reinforcement learning. *Neural Computation*, 2(4), 1990. (Also In the Proceedings of the Seventh International Conference on Machine Learning, Morgan Kaufmann, June 1990).

[52] Steven D. Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1), 1991. (Also Tech. Report # 331, Department of Computer Science, University of Rochester, 1990.).

[53] Steven D. Whitehead and Dana H. Ballard. A study of cooperative mechanisms for faster reinforcement learning. TR 365, Computer Science Dept., University of Rochester, Feburary 1991.

[54] Steven D. Whitehead, Jonas Karlsson, and Josh Tenenberg. Learning multiple goal behavior via task decomposition and dynamic policy merging. In *Robot Learning*. MIT Press, Cambridge, MA, forthcoming.

[55] Ronald J. Williams. Reinforcement learning in connectionist networks. Technical Report Technical Report ICS 8605, Institute for Cognitive Science, University of California at SanDiego, 1986.

[56] Ronald J. Williams. Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3, College of Computer Science, Northeastern University, Boston, MA, 1987.

[57] Lambert E. Wixson and Dana H. Ballard. Learning efficient sensing sequences for object search. In *AAAI Fall Symposium*, November 1991.

[58] A.L. Yarbus. *Eye Movements and Vision*. Plenum Press, 1967.

[59] Richard C. Yee, Sharad Saxena, Paul E. Utgoff, and Andrew G. Barto. Explaining temporal-differences to create useful concepts for evaluating states. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1990.