ELSEVIER

Artificial Intelligence 73 (1995) 271-306

Artificial Intelligence

Reinforcement learning of non-Markov decision processes

Steven D.Whitehead [a,*], Long-Ji Lin [b,1]

[a] *GTE Laboratories Incorporated, 40 Sylvan Road, Waltham, MA 02254, USA*

[b] *School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

Received September 1992; revised April 1993

Abstract

Techniques based on reinforcement learning (RL) have been used to build systems that learn to perform nontrivial sequential decision tasks. To date, most of this work has focused on learning tasks that can be described as Markov decision processes. While this formalism is useful for modeling a wide range of control problems, there are important tasks 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 immediate sensation. 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. Existing RL algorithms perform unreliably on hidden state tasks.

This article examines two general approaches to extending reinforcement learning to hidden state tasks. The *Consistent Representation (CR) Method* unifies recent approaches such as the Lion algorithm, the G-algorithm, and CS-QL. The method is useful for learning tasks that require the agent to control its sensory inputs. However, it assumes that, by appropriate control of perception, the external states can be identified at each point in time from the immediate sensory inputs. A second, more general set of algorithms in which the agent maintains internal state over time is also considered. These *stored-state* 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 these methods are considered and conditions for their useful application are discussed.

* Correponding author. Fax: (617) 890-9320. E-mail: swhitehead@gte.com.

[1] Currently at Siemens Corporate Research Incorporated, 755 College Road East, Princeton, NJ 08540, USA. Fax: (609) 734-6565. E-mail: ljl@learning.scr.siemens.com.

0004-3702/95/$09.50 © 1995 Elsevier Science B.V. All rights reserved
SSDI 0004-3702(94)00012-P

1. Introduction

Computational theories of agent–environment interaction need to include an account of learning. Learning is necessary to develop and maintain intelligent agents. Sophisticated real-world robots cannot be achieved through meticulous programming alone. The real world is much too complex, idiosyncratic, and uncertain to know ahead of time, and programming languages are too rigid and inexpressive for programming alone to be feasible. Intelligent agents must bear at least some of the burden of skill acquisition themselves. Also, because the world does not stand still, to maintain a high level of performance, agents must learn new skills and adapt old ones to changes in the environment.

Though there are many kinds of learning, and many things that an agent might learn, in the end, all learning boils down to learning control. The value of anything learned can only be measured in terms of its effect on the agent's interaction with its environment, in terms of its ability to control the environment to a desired end.

This article focuses on reinforcement learning, a paradigm that is well suited to learning control in highly interactive environments [7,46]. In reinforcement learning the agent–environment interaction is modeled as a controller coupled to a finite state machine (whose transition probabilities are unknown). At each time step, the controller performs an action which causes the environment to change state and generate a payoff. The agent's objective is to learn a state-dependent control policy that maximizes a measure of the total payoff received over time.

Some of the features that make reinforcement learning (RL) appealing are:

(1) RL is a weak method in that
   (a) learning occurs through trial-and-error experimentation with the environment;
   (b) the feedback used for learning takes the form of a scalar payoff—no explicit teacher, who offers the "correct answer" is required;
   (c) on sequential decision making tasks, payoffs may be sparse and considerably delayed; and
   (d) little or no prior knowledge is required.
(2) RL is incremental and can be used online.
(3) RL can be used to learn direct sensory–motor mappings, and is, thus, appropriate for highly reactive tasks in which the agent must respond quickly to unexpected events in the environment.
(4) RL is valid in nondeterministic environments.
(5) When used in conjunction with temporal difference (TD) methods [19,44], RL has proven to be effective on difficult sequential decision making tasks (e.g., checkers [38], pole balancing [8,30], and backgammon [49]).
(6) RL architectures are extensible. Recently, RL systems have been extended to incorporate aspects of planning [25,45,58], intelligent exploration [21,23,51], supervised learning [15,24,56] and hierarchical control [16,26,41,62].

Traditionally, research in RL has focused on Markov decision processes (MDPs). Described formally in Section 2, a Markov decision process intuitively corresponds to a control task in which at each point in time (a) the agent directly observes the

state of the environment and (b) the effects of actions depend only upon the action and the current state. In most applications, the agent does not observe the state of the environment directly, in the sense of being given a label which names the state. This assumption that the agent, at each point in time, encodes in its internal representation all the information relevant to predicting the effects of actions is known as the *Markov* assumption. It is important for two reasons. First, it has been important to the theoretical development of RL, because focusing on Markov decision processes has allowed researchers to apply the classical mathematicals of stochastic processes and dynamic programming [7,44,54,55]. Second, existing reinforcement learning methods, which use TD-methods [44], rely on the Markov property during credit assignment, and may perform badly when the assumption is violated [18,59]. Nevertheless, there are important learning 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 they occur whenever it is possible for a relevant piece of information to be *hidden* (or missing) from the agent's representation of the current situation.

Hidden state tasks arise naturally in the context of autonomous learning robots. The simplest example of a hidden state task is one which occurs when the agent's sensors are inadequate for the task at hand. Suppose a robot is charged with the task of sorting blocks into bins according to their color, say Bin-1 for red, Bin-2 for blue. If the robot's sensors are unable to distinguish red from blue, then for any given block it can do no better than guess a classification. If there are an equal number of blocks of each color, then guessing can do no better than chance. On the other hand, if the robot can detect color, it can easily learn to achieve 100% performance. The former case corresponds to a non-Markov decision problem, since relevant information is missing from the agent's representation. The latter case is Markov since once a color sense is available the information needed to achieve optimal performance is always available. In general, if a robot's internal representation is defined only 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 is non-Markov.

Hidden state tasks are also a natural consequence of integrating learning with active/selective perception [60]. In active perception, the agent has a degree of control over the allocation of its sensory resources (e.g., controlling visual attention or selecting visual processing modules) [3,5,6]. 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 fail to code a relevant piece of information, and the resultant internal representation may be ambiguous with respect to the current state of the environment. It follows that if the agent must learn to control its sensors, there will be periods of time when 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 [57]. The principal idea underlying the CR-method is to split con-

trol 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 (i.e., Markov) representation of the current external environment. During the overt stage, this representation is used to select overt action; that is, an action that changes 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. Although, the CR-method unifies such recent algorithms as the Lion algorithm [60], CS-QL [48], and the G-algorithm [13], it is restricted to tasks in which the agent can always identify the current state of the environment through proper control of its sensors. It is not appropriate for tasks in which the agent must store state information over time, or remember previous events in order to infer the current state.

To overcome this restriction, several, more general approaches are considered that retain state information over time. We refer to these as *stored-state methods*. The simplest of these approaches is one which augments immediate sensory inputs with a delay line to achieve a crude form of short term memory [25]. This approach has been successful in certain speech recognition tasks [53]. Another alternative is called the method of *predictive distinctions* [4, 14, 25, 39]. Following this approach, the system learns a predictive model of the sensory inputs (i.e., environmental observables) and then uses the internal state of this model to drive action selection. A third approach uses a recurrent neural network in combination with existing reinforcement learning methods to learn a recurrent state-dependent control policy directly [28]. Each of these methods is described in detail and conditions for their useful application are considered.

The remainder of the article is organized as follows. Section 2 provides a basic review of concepts from reinforcement learning. Section 3 discusses non-Markov decision processes and considers the difficulties they cause for learning control. Section 4 presents the Consistent Representation Method as a technique for learning to control perception and reviews examples of this technique. Section 5 describes and compares three stored-state methods and specifies preference conditions 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 reinforcement learning

In this section the basic concepts from reinforcement learning are reviewed. We begin by describing a simple model of agent–environment interaction. Next, we review the principles of Markov decision processes and Q-learning [54], a popular reinforcement learning algorithm. A thorough review of Markov decision processes and reinforcement learning, however, is beyond the scope of this article. Throughout the article, we shall focus primarily on Q-learning and the difficulties caused for it by non-Markov decision processes. Other algorithms [8, 19, 44, 63] suffer a similar fate. For a more complete review of Markov decision processes and Q-learning, the reader may wish to consult [10] and [54]. For a review of reinforcement learning in general, see [7].

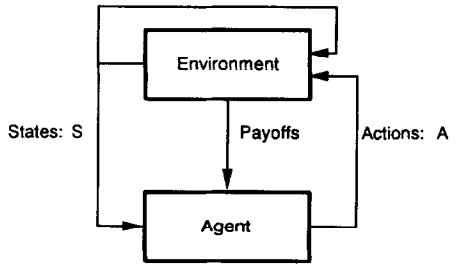

Fig. 1. A simple model of agent–environment interaction.

2.1. Modeling agent–environment interaction

Fig. 1 illustrates a model of agent–environment interaction that is widely used in reinforcement learning. 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 time point, the following sequence of events occurs.

(1) The agent senses the state of the environment.
(2) Based on this current state, the agent chooses an action to perform.
(3) Based on the current state and the action selected by the agent, the environment makes a transition to a new state and generates a payoff.
(4) The payoff is passed back to the agent.

2.1.1. The environment

The environment is modeled as a Markov decision process. Formally, a Markov decision process is described by the tuple (S, A, T, R), where S is the set of possible states, A is the set of possible actions, T is the *state transition function*, and R is the *reward function*. At each time, the environment occupies exactly one state from S, and accepts one action from A. S and A are usually assumed to be discrete and finite. State transitions are modeled by the transition function, T, which maps state–action pairs into new states ($T : S \times A \rightarrow S$). The transition function is in general probabilistic, and typically specified in terms of a set of *transition probabilities*, $P_{x,y}(a)$, where

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

Payoffs generated by the environment are determined by a reward function, R, which maps state–action pairs into scalar-valued rewards ($R : S \times A \rightarrow \mathbb{R}$). The reward function may also be probabilistic.

Notice that in a Markov decision process (MDP), the effects of actions (i.e., the next state and the immediate reward generated) depend only upon the current state. Process models of this type are said to be memoryless and to satisfy the *Markov property*. The Markov property is fundamental to this model of the environment because it implies that knowledge of the current state is always sufficient for optimal control (i.e., to maximize the reward received over time) [10]. Thus, even though it may be possible to devise action-selection strategies whose decisions depend upon additional information (e.g., a

history trace), 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 is responsible for generating control actions. At each time step it senses the current state, selects an action, and observes the new state and reward that result. Rewards are used as feedback for learning.

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 that maximizes the *expected return*.

For a fixed policy f, define $V_f(x)$, the *value function* for policy 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 is uniformly best for all possible states. That is, find f^*, such that

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

An important property of MDPs is that f^* is well defined and guaranteed to exist. In particular, the *Optimality Theorem* from dynamic programming [9] 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:

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

where $Q_f(x,a)$, the *action-value function*, is defined to be the expected return given that the agent starts in state x, applies action a once, and follows policy f thereafter [9, 10]. Intuitively, Eq. (4) states that a policy is optimal if and only if in each state, the policy specifies an action that maximizes the local "action-value". That is,

$$f^*(x) = a \text{ such that } Q_{f^*}(x,a) = \max_{b \in A}[Q_{f^*}(x,b)] \quad \forall x \in S, \tag{5}$$

and

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

$Q \leftarrow$ a set of initial values for the action-value function (e.g., uniformly zero)
Repeat forever:

1) $x \leftarrow$ the current state.
2) Select an action a to execute that is usually consistent with $f(x)$ but occasionally an alternate.
3) Execute action a, and let y be the next state and r be the reward received.
4) Update $Q(x,a)$:
$$Q(x,a) \leftarrow (1-\alpha)Q(x,a) + \alpha[r + \gamma U(y)]$$
where $U(y) = Q(y, f(y))$.

Here for each $x \in S$: $f(x) \leftarrow a$ such that $Q(x,a) = \max_{b \in A} Q(x,b)$.

Fig. 2. A simple version of the one-step Q-learning algorithm.

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

If an MDP is completely known (including the transition probabilities and reward distributions), then the optimal policy can be computed directly using techniques from dynamic programming [9, 10, 36]. However, in many cases, the structure and dynamics of the environment are *not* known. 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 [54] is an incremental reinforcement learning method. It is a good representative for reinforcement learning because it is simple, 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 [44]), an understanding of the difficulties for Q-learning is useful for understanding weaknesses of other algorithms.

In Q-learning the agent estimates the optimal action-value function (also called the Q-function) directly and uses it to derive a control policy using the greedy strategy mandated by Eq. (5). A simple Q-learning algorithm is shown in Fig. 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). After initialization, the agent enters the main control/learning loop. The first step is to sense the current state, x. Next, the agent selects an action a to execute. Most of the time, this action will be the action specified by its policy $f(x)$ as defined by Eq. (5), but for the purposes of exploration the agent will occasionally choose an action that appears sub-optimal. For example, one

might choose to follow f with probability p and choose a random action otherwise.[2] The agent executes the selected action and notes the reward r and state y that result. Finally, the action-value estimate for state-action pair (x, a) is 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

$$r + \gamma U(y), \tag{7}$$

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

$$Q^*(x, a) = E[R(x, a) + \gamma V^*(T(x, a))], \tag{8}$$

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

$$Q(x, a) \leftarrow (1 - \alpha)Q(x, a) + \alpha[r + \gamma U(y)], \tag{9}$$

where α is the learning rate. Q-learning is guaranteed to converge to an optimal policy for any finite Markov decision process if, in the limit, every state-action pair is tried infinitely often and if the learning rate decreases according to a proper schedule [55].

3. Non-Markov tasks

To see how non-Markov decision tasks arise in agent–environment interactions, one need only make a distinction between an abstract (agent-independent) specification for a task and the actual decision problem faced by a learning agent. We shall refer to these as the *external* and the *internal* decision problems, respectively, since the former is agent-independent and exists outside the agent, while the latter is endogenous and agent-specific.

3.1. An example

To illustrate the distinction, consider the task of inspecting apples in a packaging plant. Suppose apples are to be sorted according to whether or not they are ripe. Ripe apples are shipped to the supermarket, green apples are crushed for their juice. If apples come down a conveyor one at a time, then a Markov decision process model of the task might be formulated as follows. We could define a state variable called ripeness to characterize the current apple as either "ripe" or "green"—this state variable would induce a state space of size 2. The model could have two actions, which would be to either "accept" or "reject" an apple. A reward function could be defined to give a unit of reward whenever a ripe apple was accepted and a green apple was rejected, and a unit of penalty otherwise. A transition function could be defined to model the dynamics

[2] 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 [21, 45, 51].

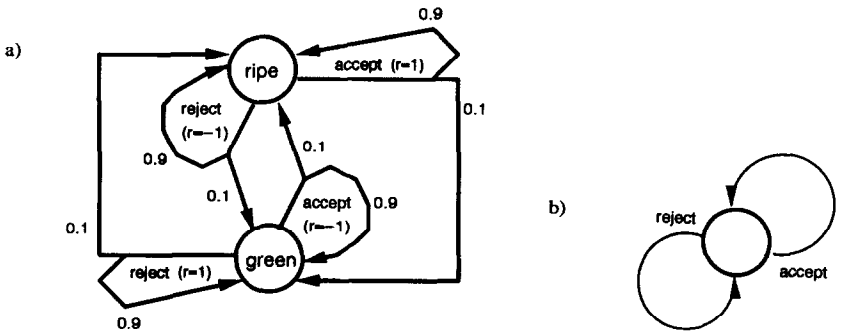

Fig. 3. The apple sorting example. (a) An abstract (external) model of the task. (b) The actual (internal) task facing a robot with no sensors.

of the apple sequence. If the types of apples coming down the conveyor are temporally dependent, as might be the case if apples were loaded into the hopper in crates from different orchards, then nonuniform transition probabilities could be used to model this dependence. For example, we might assume that the next apple down the conveyor will be the same type as the last one with probability 0.9, and different with probability 0.1.

This process model, illustrated in Fig. 3(a), corresponds to a specification of the task. It is a mathematical abstraction. The states, actions, rewards, and transitions exist in the mind of the modeler, and are intended to capture the essence of the environment and the task requirements. It makes no explicit reference to the physical agent that might perform the task. It defines the *external* decision problem.

Conversely, consider the decision problem facing a control system embedded inside a robot's head. For simplicity, consider a robot that has a single binary sensor $S1$ that it uses to represent its environment, and a single binary actuator $A1$ that it uses to affect action. In a totally situated agent, the sensors and affectors determine the basic structure of the internal decision problem: two states: $S1 = 0, S1 = 1$; and two actions: $A1 = 0, A1 = 1$. The transition and reward dynamics of this internal process are determined not only by the dynamics of the external environment, but also by the physics of the sensory–motor interface. For example, if the robot's sensor is able to detect the color of the apple on the conveyor (say, red or green), and if its affector is a lever that can be used to select apples then the internal decision process may closely match the external task (assuming red apples are "ripe" and green apples are "green"). On the other hand, if the agent's sensor or affector is not closely matched (e.g., say the sensor detects bruises), then the internal problem may bear little resemblance to the external one.

3.2. *Accounting for perceptual–motor processes*

The distinction between external and internal decision tasks can be incorporated into our model of agent–environment interaction by augmenting Fig. 1 to explicitly account for the mappings performed by the agent's perceptual–motor processes. The new model is shown in Fig. 4. On the sensory side the agent's perceptual processes map states in

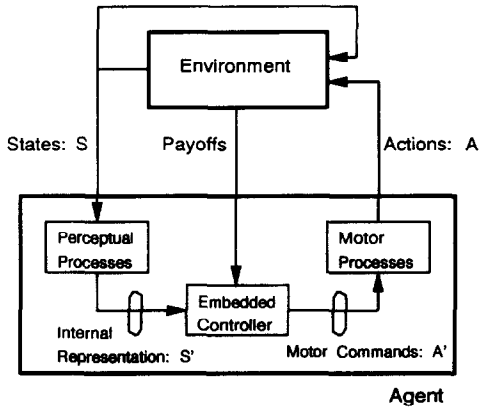

Fig. 4. A model of agent-environment interaction that explicitly accounts for the agent's perceptual-motor processes. Perceptual processes map states from the external world model onto internal representations. Motor processes map internal motor commands onto actions in the external world model.

the external world onto states in the agent's internal representation. On the motor side, the agent's motor processes map internal motor commands to actions in the external task model.[3] In general, these mappings could be of considerable complexity. A perceptual mapping might involve immediate precepts, attentional mechanisms, and stored history information; and a motor mapping might involve complex motor sequences or parallel actions. For the purposes of the present discussion we shall restrict consideration to simple mappings only. Indeed, for the remainder of the article, we will concentrate only on sensory mappings and assume that there is a one-to-one mapping between motor commands and external actions. We also assume reward is passed directly to the embedded controller. That is, whenever a situation arises in the world that would generate a reward in the external model, that reward is correctly passed to the embedded controller.

3.3. Non-Markov internal tasks

Formally, a decision task is non-Markov if information above and beyond knowledge of the current state can be used to better predict the dynamics of the process and improve control.[4]

In general, an agent's internal decision problem will be non-Markov if there are internal states that can represent multiple external states. We call this phenomenon *perceptual aliasing* since it occurs when two or more external states, through the perceptual

[3] Notice that these mappings are conceptual more than physical, in that they run through the world and then into the external (abstract) task model. For example, a state in the agent's internal representation may represent certain situations in the physical universe, which (via the modeling process) get mapped into states in the external task model.

[4] Strictly speaking, this definition is slightly overstated. Traditionally a process is said to be non-Markov if information about the *history* of the process (in addition to its current state) can be used to improve prediction and control. However, because we are dealing with systems that have the potential to collect additional information through sensing we have chosen to adopt a definition that is slightly more general.

mapping, get superimposed onto a single internal state. Intuitively, perceptual aliasing occurs when the agent is uncertain about the state of the external world. For example, in a situated agent, it occurs when the agent's sensors fail to code a relevant piece of information. Perceptual aliasing results in non-Markov decision tasks because, by definition, there are states in the internal representation that do not code all the information needed to characterize the future dynamics of the task.

Returning to our apple sorting robot, suppose the robot's color sensor is temporarily disconnected (i.e., always made to read zero). In this case, the robot is unable to distinguish apples by their color and the internal representation collapses to a single state, (see Fig. 3(b)). Clearly, this decision problem is non-Markov since (1) knowledge of the current apple's color could be used to improve the systems performance—without its sensor, the robot is reduced to guessing; and (2) knowledge of the recent history of the process could be used to improve performance. For example, knowledge that the last action resulted in a positive reward could be exploited to yield a control policy that performs better than chance. This follows since according to the external model, 90 percent of the time the current apple is the same type as the previous one.

3.3.1. The ubiquity of non-Markov tasks

Markov decision tasks are an ideal. Non-Markov tasks are the norm. They are as ubiquitous as uncertainty itself. An agent that can be uncertain about the state of the external task necessarily faces an internal decision problem that is non-Markov. And sources of uncertainty abound. Sensors have physical limitations. Rarely are they perfectly matched to the task. Sensor data are typically noisy, unreliable, and full of spurious information. Sensors have limited range, and relevant objects are often occluded. Also, information can be hidden in time. The spoken word is transient and lost unless it is actively processed and stored. Short term memory too is limited and subject to deterioration. Lin [28] 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. Such 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.

Even if perfect sensors were available, many control problems are too ambiguous or ill-posed to specify a state space in advance. Indeed, part of the agent's task may be to discover a useful state space for the problem. For example, integrating learning and active perception invariably leads to non-Markov decision tasks [57]. Active perception refers to the idea that an intelligent agent should actively control its sensors in order to sense and represent only information that is relevant to its immediate ongoing activity [1,3,5,6,12,52]. If an agent must, as part of the task, learn to control its sensors,

its internal control problem will necessarily be non-Markov. This follows since during learning there will be periods of time when the agent will improperly control its sensors, fail to attend to a relevant piece of information, and fail to unambiguously identify the state of the external task.

3.4. *Effects on control*

The level of performance that can be obtained by an agent whose internal decision problem is non-Markov is generally inferior to that of its Markov counterpart. That is, a non-Markov decision problem usually leads to sub-optimal control if the agent mistakenly assumes the problem is Markov. This can be seen in the apple sorting task, where the robot with its color sensor intact can achieve perfect classification, but the robot without the sensor (and no memory) can perform no better than chance. Notice that the degree of performance degradation depends on the problem. For example, if 95% of the apples are ripe, then a blind policy that "accepted" every apple would fail only 5% of the time. On the other hand, perceptual aliasing can lead to non-Markov decision tasks whose best fixed policies are arbitrarily bad.[5]

3.5. *Difficulties for reinforcement learning*

The straightforward application of traditional reinforcement learning methods to non-Markov decision problems in many cases yields sub-optimal control and in some cases severely degraded performance. These difficulties stem from the agent's inability to obtain accurate estimates of the utility and action-values for the underlying decision process. In particular, because of perceptual aliasing, there are states in the agent's internal representation that represent two or more distinct states in the external world model. The utility and action-value estimates learned by the agent for these non-Markov internal states tend to reflect a mix (or average) of the values for the external states they represent. Because of this averaging, it is inevitable that the agent's internal utility and action-value estimates will be erroneous for some situations. These errors, in turn, may result in the selection of sub-optimal actions for the non-Markov states, as the agent may inaccurately perceive a sub-optimal action to be of higher value than the true optimal action, or the agent may incorrectly degrade the value of the optimal action.

Unfortunately these difficulties are compounded by use of temporal difference methods [44] which cause errors to propagate throughout the state space, thus infecting action selection even for non-aliased states. In particular, most reinforcement learning algorithms employ utility (action-value) estimators that combine values for recently observed rewards with utility estimates for subsequent states—in one-step Q-learning, action-value estimates are obtained by adding the immediate reward with a utility estimate for the next state (cf. Eq. (7)). Thus, for a given state if the agent constructs estimators that use utility estimates from non-Markov states, those estimators will likely

[5] This is significant because most reinforcement learning algorithms aim to learn fixed policies. Opening the door to probabilistic policies could improve performance under these circumstances.

be erroneous and the error will be propagated to that state. Once infected a state may propagate its error to other states in a similar manner.

4. Consistent representation methods

In the last few years several RL algorithms have been developed to deal with selective perception. The Lion algorithm [60] learns to control visual attention in a primitive deictic sensory–motor system; the CS-QL algorithm [48] learns efficient, task-specific sensing operations; and the G-algorithm [13] learns to extract task-relevant bits from a large input vector.[6] In this section, we review these algorithms in turn. We then present the *Consistent Representation Method*, a computational framework that unifies these different algorithms [62].

4.1. The Lion algorithm

The Lion algorithm was perhaps the first reinforcement learning algorithm specifically designed to address an adaptive perception task [59]. 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 controllable sensory 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 block-stacking 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. 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 Block 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

[6] Note that these methods differ from supervised feature selection methods [35] that rely on the presentation of preclassified samples. The algorithms presented here operate without explicit supervision in the context of an embedded reinforcement learning task.

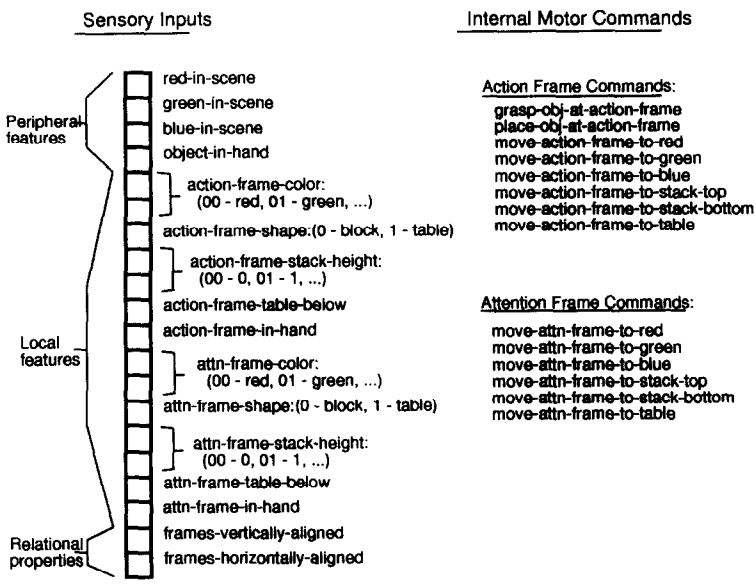

Fig. 5. A specification for the deictic sensory–motor system used by Meliora-II (Whitehead [60]). The system has two markers, an *action-frame* marker and an *attention-frame* marker. The system has a 20-bit input vector, 8 overt actions, and 6 perceptual actions. The values registered in the input vector and the effects of internal action commands depend upon the bindings between markers in the sensory–motor system and objects in the environment.

markers [1,52]. Conceptually, a marker corresponds to a focus of attention. In practice, markers are used to establish frames of reference for both perception and 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 Fig. 5. This system employs two markers: the action-frame marker and the attention-frame marker, respectively. 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 onto its relatively small input register.

Listed on the right-hand side of Fig. 5 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-

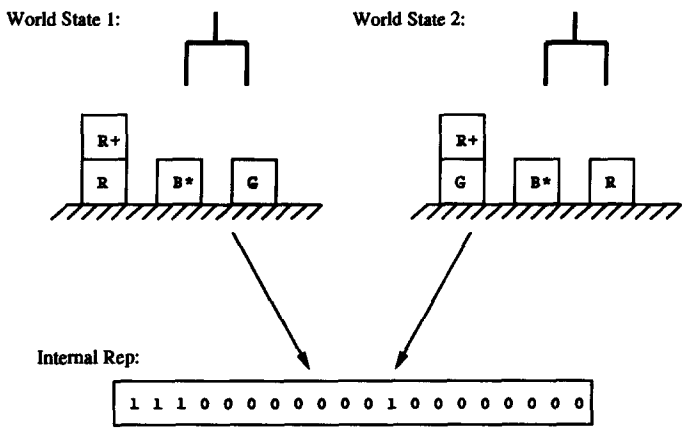

Fig. 6. An example of perceptual aliasing in the block-stacking domain. In this case, two world states with different utilities and optimal actions generate the same internal representation. The "*" indicates the location of the attention-frame marker; the "+" the location of the action-frame.

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 Fig. 6 which shows two different external world states (each corresponding to a different state in a Markov model of the task) that, because of an improper placement of markers, 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 unaliased (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 is 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 determining 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.[7] To guide

[7] 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.

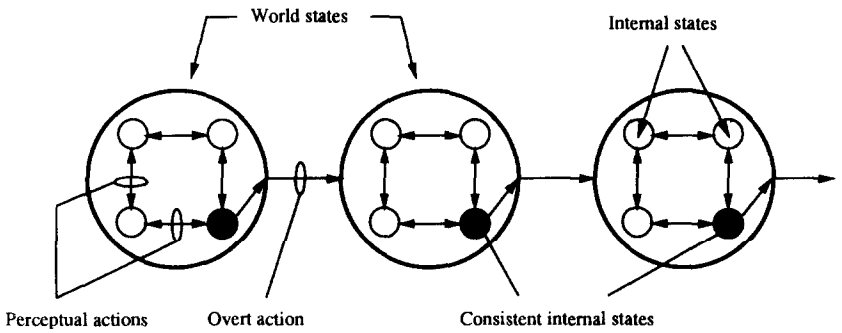

Fig. 7. A graphical depiction of the Lion algorithm. The large (super) graph depicts the overt control cycle, where large nodes correspond to world states and arcs correspond to overt actions. The subgraphs embedded within each large node depict perceptual cycles, with nodes corresponding to internal representations of the current world state and arcs corresponding to perceptual actions.

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 special in that the action-values for perceptually aliased states are suppressed (i.e., ideally they are equal to zero), whereas the action-values for unaliased 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 with the maximum action-value. Since aliased states tend to have suppressed action-values, the selected action tends to correspond to the optimal action from a unaliased internal state. Fig. 7 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 one-step Q-learning (i.e., Eq. (9)). 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 an unaliased state, further changes in the action-values for aliased states cease. Finally, each buffered state is tested to see if it is aliased. If a state tests positive, its action-value is reset to zero.

A very simple procedure is used to identify potentially aliased states. The rule simply examines the sign of the error term in the one-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 set to zero, the task is deterministic, and rewards are nonnegative, then aliased states tend to regularly overestimate their action-values (i.e., show a negative error). Unaliased states, on the other hand, tend to monotonically approach the optimal action-value from below (i.e., positive error only). Therefore, aliased states can be detected by monitoring the sign in

the estimation error.[8]

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, actions are selected by choosing the perceptual 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 one-step Q-learning rule except that the overt utility of the internal state is also accounted for. Since aliased states tend to have suppressed overt action-values, perceptual actions that lead to unaliased internal states tend to have higher action-values than those that do not. (See [57] 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 [57].

The Lion algorithm exploits several assumptions in order to deal with non-Markov decision problems. These limiting assumptions are:

(1) The effects of actions must be deterministic.
(2) Only nonnegative 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 unaliased.

4.2. CS-QL

There is much work in machine learning that focuses upon the predictive power of information, but fails to account for its cost [2, 35]. Tan recognized that to be efficient it is necessary to explicitly account for the cost of sensing. In [48], he develops two cost-sensitive learning algorithms for classification tasks: CS-ID3 and CS-IBL, respectively. CS-QL, which stands for Cost-Sensitive Q-Learning, resulted when he combined ideas for cost-sensitive learning with reinforcement learning [47]. In CS-QL, the learning agent not only learns the overt actions needed to perform a task, but also learns an efficient procedure for identifying the current state of the environment.

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 has a set of primitive sensing tests. Each test provides a specific piece of information about the external environment. Also, instead of learning a perceptual control policy, as in the Lion algorithm, CS-QL constructs a classification tree,

[8] Subtle interactions sometimes cause unaliased states to overestimate their action-values. This sometimes leads to suppression of unaliased states. However these states tend to bounce back from such suppressions and eventually stabilize. For a detailed discussion of this technique for detecting aliased states, see [57] and [48].

where each internal node corresponds to a sensing operation, each branch corresponds to a test result, and each leaf corresponds to a state in the agent's internal representation (for example, see Fig. 8). In CS-QL, the agent has learned an adequate classification tree when every leaf in the tree is unaliased; 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. In other words, the entire external state space is collapsed onto a single internal state. As aliased 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 aliased leafs, CS-QL uses the same technique employed by the Lion algorithm. Thus, CS-QL shares the same limitations as the Lion algorithm.

CS-QL has been successfully demonstrated in a simulated robot navigation task in which the robot must learn to deduce its position from limited information gathered from its selective sensors. One very simple instance of this type of navigation task, along with a classification tree learned by CS-QL, is shown in Fig. 8. In this example, 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 proportional to its distance from the robot. By accumulating features from nearby cells the robot can successfully (and efficiently) identify its position within the maze.

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 [13] 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 that their learning system was being overwhelmed by the sheer volume of information generated by the sensors. The subtask they were studying involved aligning the agent with a target, orienting to it, and firing a weapon. At each point in time, the agent's sensory system generated 100 bits of input. Using all this information resulted in an internal state space containing 2^{100} states. Most of these bits, however, were irrelevant to *the* specific subtask they were studying 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

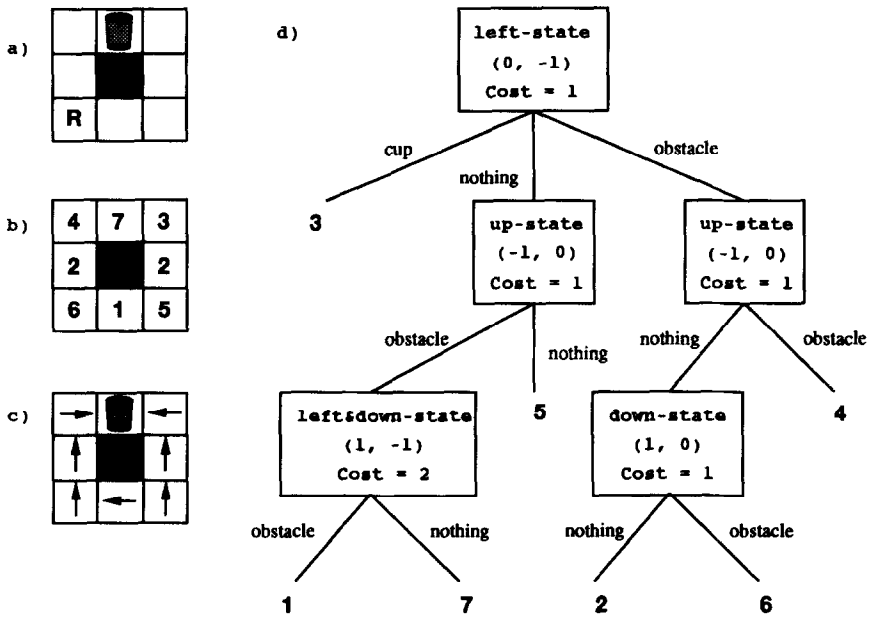

Fig. 8. A simple example of CS-QL: (a) a 3 × 3 grid world, (b) a learned mapping between state descriptions and states, (c) a learned optimal decision policy, and (d) a learned cost-sensitive classification tree. (Reproduced with permission.)

other stages of the game, the bits that were irrelevant "now" were vitally important "then". The G-algorithm was developed to learn a control policy which could generalize over irrelevant information in the input.

The G-algorithm works by identifying bits in the input vector that are important for control. It is very similar to CS-QL in that both grow classification trees incrementally. That is, both start with a single root node (i.e., assuming no information is relevant), then construct a tree-structured 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 (aliased) 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 that there are bits in the input vector (that have not already been tested in traversing the tree from root-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 [42] to find statistically significant bits. That is, over time as the agent experiences a

variety of external states, reward data is collected and stored for each leaf, target-bit pair. Data for a given leaf is separated into one of two bins. One bin corresponds to situations when the target bit is on; the other when the bit is off. 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 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 reward 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. 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. 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 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 aims to generate optimal overt actions.
(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 absolutely necessary for the internal state space to be Markov. In particular, it is sufficient for each state to be Markov with respect to predicting future rewards (but not necessarily future states). For instance, in Fig. 8, two different external states are classified into the same state, state 2. This "mis-classification", however, does not result in a sub-optimal policy. This slightly weaker concept of being "partially Markov" or "Markov with respect to reward" has been associated with the term "consistent". See [57] for a further discussion of this distinction.

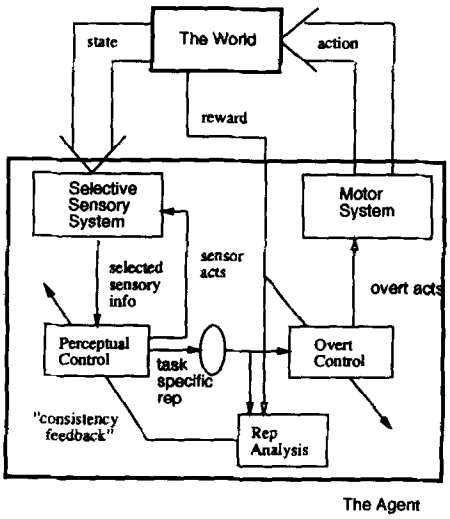

Fig. 9. The basic architecture of a system using the Consistent Representation Method. Control is accomplished in two stages: a perceptual stage, followed by an overt stage. The goal of the perceptual stage is to generate a Markov, task-dependent internal state space. The goal of overt control is to 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.

Fig. 9 illustrates an architectural embodiment of the CR-method. The major components include: a selective sensory system, a motor system, a perceptual control, an overt control, and a representation monitor. The line from the perceptual control to the selective sensory system represents perceptual control (or selection) acts. The line from the overt control to the motor system represents overt acts. Both the perceptual and overt controllers are adaptive. Reward from the environment is received by both the overt controller and the representation monitor. The representation monitor detects non-Markov states and provides feedback to the perceptual control.

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 as relevant. The identification procedure implemented in the perceptual control 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-algorithm 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-algorithm 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 stored-state approaches to them are the subject of the next section.

5. Stored-state methods

One obvious approach to dealing with inadequate perception and non-Markov decision problems is 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 [4, 14, 28, 39, 50]. In addition to these two approaches, this section describes a third approach, which learns a history-sensitive control policy directly from reinforcement.

5.1. Three stored-state architectures

Fig. 10 depicts three stored-state architectures for reinforcement learning in non-Markov domains. In our experiments with all three, a neural network (Q-net) was 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, the agent 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

[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 [57].

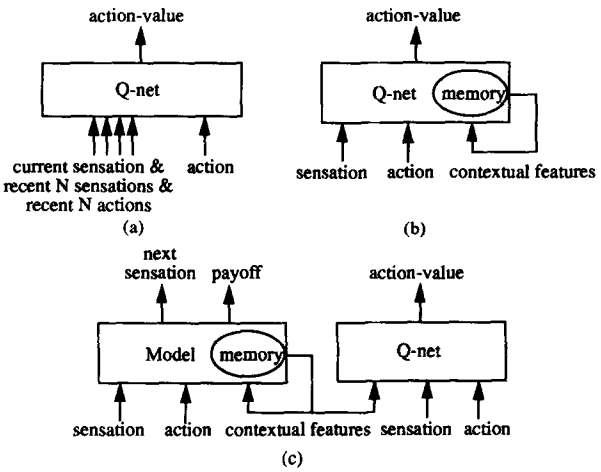

Fig. 10. Three stored-state architectures for reinforcement learning in non-Markov domains: (a) window-Q, (b) recurrent-Q, and (c) recurrent-model.

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 *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-delay neural network* has been found to be useful in speech recognition tasks [53], 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 contextual features are constructed correctly then the resultant internal state space will be Markov and standard RL methods may be used to learn an optimal control policy. The *recurrent-Q* and *recurrent-model* architectures illustrated in Fig. 10 are based on this basic idea. However, they differ in the way they construct their contextual features. Unlike the window-Q architecture, both of these architectures can in principle discover and utilize historical information that depends on sensations arbitrarily deep in the past, although in practice this has been difficult to achieve.

Recurrent neural networks, such as Elman networks [17], provide one approach to constructing relevant contextual features. As illustrated in Fig. 11, 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

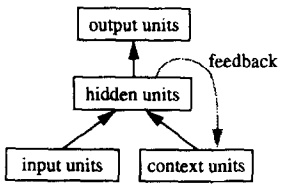

Fig. 11. An Elman network.

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 (Fig. 10(b)). To predict action-values correctly, the recurrent network (called recurrent Q-net) must learn contextual features which enable the network to distinguish between different external states that generate the same immediate sensory inputs.

The recurrent-model architecture (Fig. 10(c)) consists of two concurrent learning components: a *one-step prediction module* or action model, and a Q-learning module. The prediction module is responsible for learning to predict the immediate sensory inputs (and payoffs) 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 contextual features to accurately predict the effects of an action on the the environment. If we assume that an accurate predictive model can be learned, and that the model's contextual 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 contextual 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.

Both the recurrent-Q and recurrent-model architectures learn contextual 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 immediate 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-values (see Eq. (9)). 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

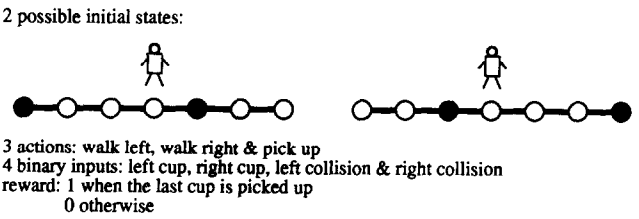

Fig. 12. Task 1: a two-cup collection task. The cup locations are denoted by filled circles.

the model network and the Q-network such that the contextual features learned would be 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 (nonrecurrent) Q-nets of the window-Q and recurrent-model architectures can be trained using a straightforward combination of temporal difference methods [44] and the connectionist back-propagation algorithm [37]. This combination has been successfully applied to solve several nontrivial reinforcement learning problems [24, 25, 49].

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* [37]. 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 [28].

5.3. *Simulation results*

This subsection presents simulation results of a study in which the three history-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 [28].)

5.3.1. *Task 1: two-cup collection*

We begin with a simple two-cup collection task (Fig. 12). This task requires the learning agent to pick up two cups located in a 1-D space. The agent has three 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 four binary bits: two bits indicating if

there is a cup in the immediate left or right cell, and two 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 cup collection problem is restricted so that there are only two possible initial states (Fig. 12). In each trial, the agent starts with one of the two initial states. Because the agent can see only one of the two cups at the beginning of each trial, the location of the other cup can only be learned from previous trials. To collect the cups optimally, the agent must use contextual information, such as which initial state it starts with, to decide which way to go after picking up the first cup. Note that the reason for restricting the possible initial states is to avoid perceptual aliasing at the onset of a trial when no contextual information is available.

This task is nontrivial for several reasons: (1) The agent cannot sense the cup in its current cell, (2) it gets no reward until both cups are picked up, and (3) it often operates with no cup in sight especially after picking up the first cup.

The three memory architectures were tested on this cup collection task. The experiment was repeated 5 times, and every time each architecture 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.

The two main results of this experiment were:

- To demonstrate that each of the architectures is capable of learning to perform this simple hidden state task, and in particular, to demonstrate that the two recurrent architectures are able to develop and use memory-based contextual features.
- To notice that for the recurrent-model architecture, even when the model is only partially correct it may provide sufficient contextual 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 sensory input. 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 had little impact on the performance of the window-Q architecture or the recurrent-Q architecture, but had a noticeable negative impact on the recurrent-model architecture.

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

In three out of the five runs, the window-Q architecture successfully found the optimal policy, while in the other two runs, it only found sub-optimal 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 contextual 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 to apply forces to the base of a movable cart in order to balance a pole that is attached to the cart via a hinge (Fig. 13). 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

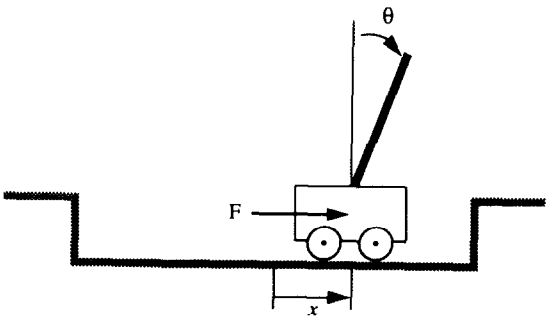

Fig. 13. The pole balancing problem.

reinforcement signals. In particular, in most formulations of the problem, the system only receives nonzero 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 [43]. 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 seven test trials where the pole starts with an angle of 0, ± 1, ± 2, or ± 3 degrees. In the training phase, pole angles and cart positions were generated randomly. The initial cart velocity and pole velocity were 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 angles and cart positions. 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 five out of six runs. (A satisfactory policy was not always found within 1000 trials.).

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

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. Comparisons

The above experiments provide some insight into the performance of the three memory architectures. This subsection considers problem characteristics that may be useful in determining when one architecture may be preferred over another.

5.4.1. Problem parameters

Some of the features (or parameters) of a task that affect 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 to generate an internal representation that is Markov. For example, the memory depth for the pole balancing task is 1, as evidenced by the fact that the window-Q agent was able to obtain satisfactory control based only on a window of size 1. Note that learning an optimal policy may require a larger memory depth than that needed to represent the policy. For instance, for Task 1 (cup collection) the window-Q agent was able to represent (and occasionally learn) the optimal policy based on a window of size as small as 2, but it could only reliably learn the optimal control when using a window of size 5.
- *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 contextual features to be learned.* In general, the more perceptual aliasing an agent faces, the more contextual features the agent has to discover, and the more difficult the task becomes. In general, predicting sensations (i.e., a model) requires more contextual features than predicting action-values (i.e., a Q-net), which in turn requires more contextual features than representing optimal policies. Consider Task 1 for example. Only two binary contextual 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 nor a perfect model in order to obtain an optimal policy. A Q-function needs to only assign values to actions so that their relative values are in the correct order. Similarly, a model needs to only provide sufficient features for constructing a good Q-function.

5.4.2. 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 contextual 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 effective. 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, action-value> triples) are not directly observable since the agent must estimate the training action-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 contextual features that are *relevant* to the control problem. The contextual 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 action-values but many are needed to predict completely the next state, the number of contextual 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 nonrecurrent networks. This advantage is offset by the disadvantage that the history information it can use is 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 contextual 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 contextual 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 contextual 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 [53] 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 patterns that already exist in a given sequence of speech phonemes. Whereas in reinforcement learning, the agent must look for the temporal patterns 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 patterns within the action sequence so as to improve its action selection policy.

On the other hand, we may improve the performance of the window-Q architecture by using more sophisticated time-delay neural networks (TDNN). The TDNN used here is quite primitive; it only has a fixed number of delays in the input layer. We can have delays in the hidden layer as well [20,53]. Bodenhausen and Waibel [11] describe a TDNN with adaptive time delays. Using their TDNN, window-Q may be able to determine a proper window size automatically.

6. Discussion

This paper has described learning techniques that were developed to handle tasks that involve either selective perception or state hidden in time. However, many tasks of interest in fact involve both selective perception and memory, and solutions to these tasks may require integration of both Consistent Representation Methods and stored-state methods.

One simple approach to extending the CR-method would be to extend the 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 reference-based rules for updating the history-trace in a way that would preserve relevant memories while dropping irrelevant ones.

Other architectures that combine features from both the CR-method and history-based architectures may also be very useful. For example, one problem with the CR-method as it currently stands is that it 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 selective perception and/or state hidden in time. To some extent we have been successful. However, 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 [15,25,61]. Nevertheless, more work is needed.
- *Fast/efficient credit assignment*: Credit assignment is the fundamental problem in reinforcement learning: given reward from the environment, which actions were responsible for generating that payoff and how should the system be changed to improve performance. Most reinforcement learning algorithms solve this problem by making incremental changes to the system over a long period of time. If additional knowledge about the causal structure of the environment (for example, an action model) can be made available, more efficient credit assignment methods can be developed. For example, see [25,31,33,34,45,64].
- *Generalization*: A reinforcement learning agent must generalize from its experiences. In particular, when the state space is so large that exhaustive search for optimal control is impractical, the agent must guess about new situations based on experience with similar situations. Instead of representing action-value functions (i.e., Q-function) with look-up tables, we must develop function approximators that promote useful generalization across states and actions. For example, see [13,27,29,32,47-49].
- *Hierarchical learning*: To date much of the work in reinforcement learning has focused on problems that are small compared to those facing real robotic systems. For example, a walking robot may require precise (continuous) information from dozens of sensors, and may need to control dozens of effectors. The combinatorics associated with such problems quickly overwhelm the simplest RL methods. Another source of complexity arises when agents pursue multiple goals. Hierarchical learning is approach to turning intractable problems into tractable ones: First, a complex task is decomposed into multiple elementary tasks that can be solved effectively. Then control policies that are learned for solving the elementary tasks can be integrated to solve the original complex task. Once the agent has learned to

solve one complex task, solving a new one task may be easier if the two share the same elementary subtasks. For work in this direction, see [22,26,27,40,62].

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.

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.

In this article we have presented several approaches to dealing with non-Markov decision problems. The Consistent Representation (CR) 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 aims to identify the current state of the environment; and an overt control phase, which aims to maximize reward. Three instances of this method, the Lion algorithm [60], the G-algorithm [13], and CS-QL [47], 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 the sensory system. This assumption prevents it from being applied to tasks in which relevant state information is temporarily hidden from view.

Stored-state methods are more appropriate for tasks in which information is hidden in time. Three stored-state 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 a 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.

Acknowledgements

We gratefully acknowledge the contributions made to this article by Dana Ballard, Tom Mitchell, Rich Sutton, Lonnie Chrisman, Ming Tan, Sebastian Thrun, and Rich Caruana. Thank you for sharing your thoughts and ideas, your comments and criticisms, and most of all your time and energy.

References

[1] P.E. Agre, The dynamic structure of everyday life, Ph.D. Thesis, Tech. Report No. 1085, MIT Artificial Intelligence Lab., Cambridge, MA (1988).

[2] D.W. Aha and D. Kibler, Noise-tolerant instance-based learning algorithms, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 794–799.

[3] J. Aloimonons, I. Weiss and A. Bandyopadhyay, Active vision, *Int. J. Comput. Vision* **1** (4) (1988) 333–356.

[4] J.R. Bachrach, Connectionist modeling and control of finite state environments, Ph.D. Thesis, University of Massachusetts, Department of Computer and Information Sciences, Amherst, MA (1992).

[5] R. Bajcsy and P. Allen, Sensing strategies, in: *Proceedings U.S.–France Robotics Workshop*, Philadelphia, PA (1984).

[6] D.H. Ballard, Animate vision, Technical Report 329, Department of Computer Science, University of Rochester, Rochester, NY (1990).

[7] 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).

[8] A.G. Barto, R.S. Sutton and C.W. Anderson, Neuron-like elements that can solve difficult learning control problems, *IEEE Trans. Syst. Man Cybern.* **13** (5) (1983) 834–846.

[9] R.E. Bellman, *Dynamic Programming* (Princeton University Press, Princeton, NJ, 1957).

[10] D.P. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models* (Prentice-Hall, Englewood Cliffs, NJ, 1987).

[11] U. Bodenhausen and A. Waibel, The Tempo 2 algorithm: adjusting time-delays by supervised learning, in: D.S. Touretzky, ed., *Advances in Neural Information Processing Systems* **3** (Morgan Kaufmann, San Mateo, CA, 1991).

[12] D. Chapman, *Vision, Instruction, and Action* (MIT Press, Cambridge, MA, 1993).

[13] D. Chapman and L.P. Kaelbling, Learning from delayed reinforcement in a complex domain, in: *Proceedings IJCAI-91*, Sydney, Australia (1991); also: Teleos Technical Report TR-90-11 (1990).

[14] L. Chrisman, Reinforcement learning with perceptual aliasing: the predictive distinctions approach, in: *Proceedings AAAI-92*, San Jose, CA (1992) 183–188.

[15] J. Clouse and P.E. Utgoff, A teaching method for reinforcement learning, in: *Proceedings Ninth International Conference on Machine Learning*, Aberdeen, Scotland (1992).

[16] P. Dayan and G. Hinton, Feudal reinforcement learning, in: J.E. Moody, S.J. Hanson and R.P. Lippmann, eds., *Advances in Neural Information Processing Systems* **5** (Morgan Kaufmann, San Mateo, CA, 1993).

[17] J.L. Elman, Finding structure in time, *Cogn. Sci.* **14** (1990) 179–211.

[18] J.J. Grefenstette, Credit assignment in rule discovery systems based on genetic algorithms, *Mach. Learn.* **3** (1988) 225–245.

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

[20] A.N. Jain, A connectionist learning architecture for parsing spoken language, Ph.D. Thesis, Technical Report CMU-CS-91-208, Carnegie Mellon University, School of Computer Science (1991).

[21] L.P. Kaelbling, Learning in embedded systems, Ph.D. Thesis, Stanford University, Stanford, CA (1990).

[22] L.P. Kaelbling, Hierarchical learning in stochastic domains: preliminary results, in: *Proceedings Tenth International Conference on Machine Learning* (1993).

[23] S. Koenig and R. Simmons, Complexity analysis of real-time reinforcement learning applied to finding shortest paths in deterministic domains, Technical Report CMU-CS-93-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1992).

[24] Long-Ji Lin, Programming robots using reinforcement learning and teaching, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 781–786.

[25] Long-Ji Lin, Self-improving reactive agents based on reinforcement learning, planning and teaching, *Mach. Learn.* **8** (1992) 293–321.

[26] Long-Ji Lin, Hierarchical learning of robot skills by reinforcement, in: *Proceedings 1993 IEEE International Conference on Neural Networks* (1993).

[27] Long-Ji Lin, Reinforcement learning for robots using neural networks, Ph.D. Thesis, Technical Report CMU-CS-93-103, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA (1993).

[28] Long-Ji Lin and T.M. Mitchell, Reinforcement learning with hidden states. in: *Proceedings Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats* (MIT Press, Cambridge, MA, 1993).

[29] S. Mahadevan and J.H. Connell, Scaling reinforcement learning to robotics by exploiting the subsumption architecture, in: *Proceedings Eighth International Workshop on Machine Learning*, Evanston, IL (1991).

[30] D. Michie and R.A. Chambers, 'Boxes' as a model of pattern-formation, in: C.H. Waddington, ed., *Toward a Theoretical Biology* **1**, *Prolegomena* (Edinburgh University Press, Edinburgh, 1968) 206–215.

[31] T.M. Mitchell and S.B. Thrun, Explanation-based neural network learning for robot control, in: J.E. Moody, S.J. Hanson and R.P. Lippmann, eds., *Advances in Neural Information Processing Systems* **5** (Morgan Kaufmann, San Mateo, CA, 1993).

[32] A. Moore, Variable resolution dynamic programming: efficiently learning action maps in multivariate real-values state spaces, in: *Proceedings Eighth International Conference on Machine Learning*, Evanston, IL (1991) 333–337.

[33] A.W. Moore and C.G. Atkeson, Prioritized sweeping: reinforcement learning with less data and less real time, *Mach. Learn.* **13** (1) (1993) 103–130.

[34] Jing Peng and R.J. Williams, Efficient learning and planning within the Dyna framework, in: *Proceedings Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats* (MIT Press, Cambridge, MA, 1993).

[35] J.R. Quinlan, Induction of decision trees, *Mach. Learn.* **1** (1986) 81–106.

[36] S. Ross, *Introduction to Stochastic Dynamic Programming* (Academic Press, New York, 1983).

[37] D.E. Rumelhart, G.E. Hinton and R.J. Williams, Learning internal representations by error propagation, in: D.E. Rumelhart and J.L. McClelland and the PDP Research Group, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* **1** (MIT Press, Cambridge, MA, 1986) Chapter 8.

[38] A.L. Samuel, Some studies in machine learning using the game of checkers, in: E.A. Feigenbaum and J. Feldman, eds., *Computers and Thought* (Krieger, Malabar, FL, 1963) 71–105.

[39] J. Schmidhuber, Reinforcement learning in Markovian and non-Markovian environments, in: D.S. Touretzky, ed., *Advances in Neural Information Processing Systems* **3** (Morgan Kaufmann, San Mateo, CA, 1991) 500–506.

[40] S.P. Singh, Transfer of learning across compositions of sequential tasks, in: *Proceedings Eighth International Workshop on Machine Learning*, Evanston, IL (1991) 348–352.

[41] S.P. Singh, Transfer of learning by composing solutions of elemental sequential tasks, *Mach. Learn.* **8** (1992) 323–339.

[42] G.W. Snedecor and W.G. Cochran, *Statistical Methods* (Iowa State University Press, Ames, Iowa, 1989).

[43] R.S. Sutton, Temporal credit assignment in reinforcement learning, Ph.D. Thesis, University of Massachusetts at Amherst (1984); also: COINS Tech. Report 84-02.

[44] R.S. Sutton, Learning to predict by the method of temporal differences, *Mach. Learn.* **3** (1) (1988) 9–44.

[45] R.S. Sutton, Integrating architectures for learning, planning, and reacting based on approximating dynamic programming, in: *Proceedings Seventh International Conference on Machine Learning*, Austin, TX (1990).

[46] R.S. Sutton, ed., *Reinforcement Learning* (Kluwer, Boston, MA, 1992).

[47] Ming Tan, Cost sensitive reinforcement learning for adaptive classification and control, in: *Proceedings IJCAI-91*, Sydney, Australia (1991).

[48] Ming Tan, Cost sensitive robot learning, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA (1991).

[49] G. Tesauro, Practical issues in temporal difference learning, *Mach. Learn.* **8** (1992) 257–277.

[50] S.Thrun and K. Moller, Planning with an adaptive world model, in: D.S. Touretzky, ed., *Advances in Neural Information Processing Systems* **3** (Morgan Kaufmann, San Mateo, CA, 1991).

[51] S. Thrun, Efficient exploration in reinforcement learning, Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1992).

[52] S. Ullman, Visual routines, *Cognition* **18** (1984) 97–159; also in: S. Pinker, ed., *Visual Cognition* (MIT Press, Cambridge, MA, 1985).

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

[54] C.J.C.H. Watkins, Learning from delayed rewards, Ph.D. Thesis, University of Cambridge, England (1989).

[55] C.J.C.H. Watkins and P. Dayan, Technical note: Q-learning, *Mach. Learn.* **82**, (1992) 39–46.

[56] S.D. Whitehead, Complexity and cooperation in reinforcement learning, in: *Proceedings AAAI-91*, Anaheim, CA (1991); a similar version also appears in: *Proceedings Eighth International Workshop on Machine Learning*, Evanston, IL (1991).

[57] S.D. Whitehead, Reinforcement learning for the adaptive control of perception and action, Ph.D. Thesis, Department of Computer Science, University of Rochester, Rochester, NY (1991).

[58] S.D. Whitehead and D.H. Ballard, A role for anticipation in reactive systems that learn, in: *Proceedings Sixth International Workshop on Machine Learning*, Ithaca, NY (1989).

[59] S.D. Whitehead and D.H. Ballard, Active perception and reinforcement learning, *Neural Comput.* **2** (4) (1990); also in: *Proceedings Seventh International Conference on Machine Learning*, Austin, TX (1990).

[60] S.D. Whitehead and D.H. Ballard, Learning to perceive and act by trial and error, *Mach. Learn.* **7** (1) (1991); also: Technical Report 331, Department of Computer Science, University of Rochester, Rochester, NY (1990).

[61] S.D. Whitehead and D.H. Ballard, A study of cooperative mechanisms for faster reinforcement learning, Technical Report 365, Computer Science Department, University of Rochester, Rochester, NY (1991).

[62] S.D. Whitehead, J. Karlsson and J. Tenenberg, Learning multiple goal behavior via task decomposition and dynamic policy merging, in: J.H. Connell and S. Mahadevan, eds., *Robot Learning* (MIT Press, Cambridge, MA, 1993).

[63] R.J. Williams, Reinforcement learning in connectionist networks, Technical Report ICS 8605, Institute for Cognitive Science, University of California at San Diego (1986).

[64] R.C. Yee, S. Saxena, P.E. Utgoff and A.G. Barto, Explaining temporal-differences to create useful concepts for evaluating states, in: *Proceedings AAAI-90*, Boston, MA (1990).