Module Based Reinforcement Learning for a Real Robot

Zsolt Kalmár, Csaba Szepesvári and András Lőrincz

Department Of Adaptive Systems

Research Group on Artificial Intelligence

JATE, Szeged, Hungary

May 8, 1997

Abstract

The behaviour of reinforcement learning (RL) algorithms is best understood in completely observable, finite state- and action-space, discrete-time controlled Markov-chains. Robot-learning domains, on the other hand, are inherently infinite both in time and space, and moreover they are only partially observable. Previous ways of overcoming these difficulties included state-space discretization based on a fixed set of features and the use of behaviours or function approximators to work with continuous spaces, but these approaches were limited both in their scope and practical utility. In this article we suggest a systematic method which unifies earlier solutions. The motivation of our method comes from the desire to transform the task-to-be-solved into a finite-state, discrete-time, "approximately" Markovian task, which is completely observable too. The key idea is to break up the problem into subtasks and design controllers for each of the subtasks. Then operating conditions are attached to the controllers (together the controllers and their operating conditions which are called modules) and possible additional features are designed to facilitate observability. A new discrete time-counter is introduced at the "module-level" that clicks only when a change in the value of one of the features is observed. The interactions of the controllers are analyzed (several tools are proposed for doing this) and RL is proposed to learn an optimal switching strategy that switches between the modules. The approach was tried out on a real-life robot. Several RL algorithms were compared and it was found that a model-based approach worked best. The learnt switching strategy performed equally well as a handcrafted version. Moreover, the learnt strategy seemed to exploit certain properties of the environment which could not have been seen in advance, which predicted the promising possibility that a learnt controller might overperform a handcrafted switching strategy in the future.

1 Introduction

Reinforcement learning (RL) is the process of learning the coordination of concurrent behaviours and their timing. A few years ago Markovian Decision Problems (MDPs) were proposed as the model for the analysis of RL [32, 23] and since then a mathematically well-founded theory has been constructed for a large class of RL algorithms. These algorithms are based on modifications of the two basic dynamic-programming algorithms used to solve MDPs, namely the value- and policy-iteration algorithms [31, 7, 11, 28, 24]. The RL algorithms learn via experiencing, gradually building an estimate of the optimal value-function, which is known to encompass all the knowledge needed to behave in an optimal way according to a fixed criterion, usually the expected total discounted-cost criterion. The basic limitations of all of the early theoretical results of these algorithms was that these assumed finite state- and action-spaces, and discrete-time models in which the state information too was assumed to be available for measurement. In a real-life problem however, the state- and action-spaces are infinite, usually non-discrete, time is continuous and the system's state is not measurable (i.e. with the latter property the process is only partially observable as opposed to being completely observable). Recognizing the serious drawbacks of the simple theoretical case, researchers have begun looking at the more interesting yet theoretically more difficult cases (see e.g. [13, 21, 29]). To date, however, no complete and theoretically sound solution has been found to deal with such involved problems. In fact the above-mentioned learning problem is indeed intractable owing to partial-observability. This result follows from a theorem of Littman's [12].

In this article an attempt is made to show that RL can be applied to learn real-life tasks when *a priori* knowledge is combined in some suitable way. The key to our proposed method lies in the using high-level modules along with a specification of the operating conditions for the modules and other "features", to transform the task into a finite-state & action, completely-observable task. Of course, the design of the modules and features require a fair amount of *a priori* knowledge, but this knowledge is usually readily available. In addition to this, there may be several possible ways of breaking up the task into smaller subtasks but it may be far from trivial to identify the best decomposition scheme. If all the possible decompositions are simultaneously available then RL can be used to find the best combination. In this paper we propose design principles and theoretical tools for the analysis of learning and demonstrate the success of this approach via *real-life* examples. A detailed comparison of several RL methods, such as Adaptive Dynamic Programming (ADP), Adaptive Real-Time Dynamic Programming

(ARTDP) and Q-learning is provided, having been combined with different exploration strategies.

The article is organized in the following way. In the next section (Section 2) we introduce our proposed method and discuss the motivations behind it. In it the notion of "approximately" stationary MDPs is also introduced as a useful tool for the analysis of "module-level" learning. Then in Section 3 the outcome of certain experiments using a mobile robot are presented. The relationship of our work to that of others is contrasted in Section 4, then finally our conclusions and possible directions for further research are given in Section 5.

2 Module-based Reinforcement Learning

First of all we will briefly run through Markovian Decision Problems (MDPs), a value-function approximation-based RL algorithm to learn solutions for MDPs and their associated theory. Next the concept of recursive-features and time discretization based on these features are elaborated upon. This is then followed by a sensible definition and principles of module-design together with a brief explanation of why the modular approach can prove successful in practice.

2.1 Markovian Decision Problems

RL is the process by which an agent improves its behaviour from observing its own interactions with the environment. One particularly well-studied RL scenario is that of a single agent minimizing the expected-discounted total cost in a discrete-time finite-state, finite-action environment, when the theory of MDPs can be used as the underlying mathematical model. A finite MDP is defined by the 4-tuple (S, A, p, c), where S is a finite set of states, A is a finite set of actions, p is a matrix of transition probabilities, and c is the so-called immediate cost-function. The ultimate target of learning is to identify an optimal policy. A policy is some function that tells the agent which set of actions should be chosen under which circumstances. A policy π is optimal under the *expected discounted total cost criterion* if, with respect to the space of all possible policies, π results in a minimum expected discounted total cost for all states. The optimal policy can be found by identifying the optimal value-function, defined recursively by

$$v^*(s) = \min_{a \in U(s)} \left(c(s,a) + \gamma \sum_{s'} p(s,a,s') \, v^*(s') \right)$$

for all states $s \in S$, where $c(s,a)$ is the immediate cost for taking action a from state s, γ is the discount factor, and $p(s,a,s')$ is the probability that state s' is reached from state s when action a is chosen. $U(s)$ is the set of admissible actions in state s. The policy which for each state selects the action that minimizes the right-hand-side of the above fixed-point equation constitutes an optimal policy. This yields the result that to identify an optimal policy it is sufficient just to find the optimal value-function v^*. The above simultaneous non-linear equations (non-linear because of the presence of the minimization operator), also known as the *Bellman equations* [3], can be solved by various dynamic programming methods such as the value- or policy-iteration methods [19].

RL algorithms are generalizations of the DP methods to the case when the transition probabilities and immediate costs are unknown. The class of RL algorithms of interest here can be viewed as variants of the value-iteration method: these algorithms gradually improve an estimate of the optimal value-function via learning from the interactions with the environment. There are two possible ways to learn the optimal value-function. One is to estimate the model (i.e., the transition probabilities and immediate

Initialization: Let $t = 0$, and initialize the utilized model (M_0) and the Q-function (Q_0)

Repeat Forever

1. Observe the next state s_{t+1} and reinforcement signal c_t .

2. Incorporate the new experience (s_t, a_t, s_{t+1}, c_t) into the model and into the estimate of the optimal Q-function: $(M_{t+1}, Q_{t+1}) = F_t(M_t, Q_t, (s_t, a_t, s_{t+1}, c_t))$.

3. Choose the next action to be executed based on (M_{t+1}, Q_{t+1}): $a_{t+1} = S_t(M_{t+1}, Q_{t+1}, s_{t+1})$ and execute the selected action.

4. $t := t + 1$.

Table 1: **The structure of value-function-approximation based RL algorithms.**

costs) while the other is to estimate the optimal action-values directly. The optimal action-value of an action a given a state s is defined as the total expected discounted cost of executing the action from the given state and proceeding in an optimal fashion afterwards:

$$Q^*(s, a) = c(s, a) + \gamma \sum_{s'} p(s, a, s')v^*(s'). \tag{1}$$

The general structure of value-function-approximation based RL algorithms is given in Table 1.

In the RL algorithms various models are utilized along with an update rule F_t and action-selection rule S_t. In the case of the Adaptive Real-Time Dynamic Programming (ARTDP) algorithm the model consists (M_t) of the estimates of the transition probabilities and costs, the update-rule F_t being implemented e.g. through the equations

$$p_{t+1}(s_t, a_t, s) = (1 - \frac{1}{n_t(s_t, a_t)})p_t(s_t, a_t, s) + \frac{1}{n_t(s_t, a_t)}\, \delta(s_{t+1}, s),$$

$$c_{t+1}(s_t, a_t) = (1 - \frac{1}{n_t(s_t, a_t)})c_t(s_t, a_t) + \frac{1}{n_t(s_t, a_t)}\, c_t,$$

where $\delta(i, j)$ is the Kronecker-function, $n_t(s, a)$ is the number of times the state-action pair (s, a) was visited by the process $\{(s_{t,}a_t)\}_t$ before time t plus one, and values not shown are left unchanged. Instead of the optimal Q-function, the optimal value-function is estimated and stored to spare storage space, and the Q-values are then computed by replacing the true transition probabilities, costs and the optimal value-function in Equation 1 by their estimates. An update of the estimate for the optimal value-function is implemented by an asynchronous dynamic programming algorithm using an inner-loop in Step 2 of the algorithm. In each step of this loop a subset of the states, F_j^t, is selected and the value-estimates of the states in F_j^t are updated via

$$v(s) := \min_{a \in U(s)} (c_{t+1}(s, a) + \gamma \sum_{s'} p_{t+1}(s, a, s')v(s')),$$

v being initialized to v_t at the beginning of the loop and letting $v_{t+1} = v$ at the end of the loop. Algorithms where the value of the actual state is updated are called "real-time" [2]. If in each step all the states are updated ($F_j^t = S$), and the inner loop is run until convergence is reached the resulting algorithm will be labelled Adaptive Dynamic Programming (ADP). Another popular RL algorithm is Q-learning, which does not employ a model but instead the Q-values are updated directly according to the iteration procedure [31]

$$Q_{t+1}(s_t, a_t) = (1 - \alpha_t(s_t, a_t))Q_t(s_t, a_t) + \alpha_t(s_t, a_t)(c_t + \gamma \min_a Q_t(s_{t+1}, a)),$$

where $\alpha_t(s_t, a_t) \geq 0$, $\sum_{t=1}^{\infty} \alpha_t(s, a)\delta(s, s_t)\delta(a, a_t) = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2(s, a)\delta(s, s_t)\delta(a, a_t) < \infty$. For example, one might set $\alpha_t(s, a) = \frac{1}{n_t(s,a)}$ but often in practice

$\alpha_t(s,a) = \text{const}$ is employed which while yielding increased adaptivities no longer ensures convergence.

Both algorithms mentioned previously are guaranteed to converge to the optimal value-/Q-function if each state-action pair is updated infinitely often. The action selection procedure S_t should be carefully chosen so that it fits the dynamics of the controlled process in a way that the condition is met. For example, the execution of random actions meets this "sufficient-exploration" condition when the MDP is communicating. However, if on-line performance is important then more sophisticated exploration is needed which, in addition to ensuring sufficient exploratory behaviour, exploits accumulated knowledge. If the Q-values were already exact then according to what was explained above the optimal action in state s would be $\operatorname{argmin}_a Q_t(s,a)$, the choice of such actions corresponding to *pure exploitation*. Unfortunately, pure exploitation applied from the beginning of learning will not work in general (a noteworthy exception being the worst-case cost-criterion [6]). Typical suggestions to overcome these difficulties include choosing random actions occasionally and exploiting actions at other times, or to select actions that minimize some kind of biased Q-values. The most popular of these is randomization when the exploiting action is chosen with a probability smaller than one, while having this probability converge slowly to one with time. Recently it has been shown that if the probability of selecting non-exploiting actions summed up in time equals infinity (for example, when this probability is proportional to $1/n_t(s)$, where $n_t(s)$ is the number of times the state s was visited before time t increased by 1), then on the one hand sufficient exploration is ensured while on the other the whole process eventually converges to optimality [22]. The most common form of randomized action selections is called "Boltzmann-exploration", where the probability of choosing action a in state s equals

$$\frac{e^{Q_t(s,a)/T(s,t)}}{\sum_{a\in U(s)} e^{Q_t(s,a)/T(s,t)}},$$

where $T(s,t)$ is a "temperature" parameter whose rate of decrease with t should be bounded from below by $o(1/\ln(n_t(s))$ if one wants to ensure sufficient exploration.

2.2 Recursive Features and Feature-based Time-Discretization

In case of a real-life robot-learning task the dynamics cannot be formulated exactly as a *finite* MDP, nor is the state information available for measurement. This latter restriction is modelled by Partially-Observable MDPs (POMDPs) where (in the simplest case) one extends an MDP with an *observation function* h which maps the set of states S into a set X, called the observation set (which is usually non-countable, just like S). The defining assumption of a POMDP is that the full state s can be observed only through the observation function, i.e. only $h(s)$ is available as input and this information alone is usually insufficient for efficient control since h is usually a non-injection (i.e. h may map different states to the same observations). *Features* which mathematically are just well-designed observation functions, are known to be efficient in dealing with the problem of infinite state-spaces. Moreover, when their definitions are extended in a sensible way they become efficient in dealing with partial observability.

It is well known that POMDPs optimal policies can depend on their whole past histories. This leads us onto a mere generalization of features, such that the feature's values can depend on all past observations, i.e. mathematically a feature becomes an infinite sequence of mappings $(f_0, f_1, \ldots f_t, \ldots)$, with $f_t : (X \times A)^t \times X \to F$, where F and X are the feature- and observation-spaces.

Since RL is supposed to work on the output of features, and RL requires finite spaces it means that F should be finite. Features that require infinite memory are clearly impossible to implement, so features used in practice should be restricted in such a way that they require a finite "memory". For example, besides stationary features which take the form $(f_0, f_0, \ldots, f_t = f_0, \ldots)$ and are called *sensor-based* features, *recursive* features (in control theory these are called *filters* or *state-estimators*) are those that can be implemented using a finite memory[1]. For example, in the case of a one-depth recursive feature the value at the t^{th} step is given by $f_t = R(x_t, a_{t-1}, f_{t-1})$, where $R : X \times A \times F \to F$ defines the recursion and $f_0 = f_0(x_0)$ for some function $f_0 : X \to F$[2]. Features whose values depend on the past-observations of a finite window form a special class of recursive filters. A very simple one-step recursive feature is the switching-feature (of Boolean type) whose value depends on two disjoint sets of observation-action pairs, those of "on-pairs" and "off-pairs". The feature's value is one (or `'on'`), if the last observed observation-action pair which was either an on-pair or off-pair is an on-pair, otherwise it is zero (`'off'`). In other words, the feature's value does not change as long as the observation is outside the union of the sets of "on-pairs" and "off-pairs" and the feature's value is reset to the label of these sets when the observation data gets into either of them. Later in the application-section we will give an example of such a feature.

Instead of relying on a single feature, it is usually more convenient to define

[1] If the state space is infinite then not all sensor-based features can be realized in practice.

[2] If time is continuous then recursive features should be replaced by features that admit continuous-time dynamics such as $\dot{f} = R(x, a, f)$. In order to have a finite-space output such features should be used together with discretization-mappings, i.e. the output of the feature is given by $h_R(f)$ instead of f, where $h_R : F_0 \to F$ is a discretization-mapping, F_0 being a suitable subset of a vector-space (such as a connected subset of $\Re$) and F the finite feature-space.

and employ a set of features, each of which indicates a certain event of interest. Note that a finite set of features can always be replaced by a single feature whose output-space is the Cartesian product of the output spaces of the individual features and whose values can be computed componentwise by the individual single features. That is to say, the new feature's values are the 'concatenated' values of the individual features.

Since the feature-space is finite, a natural discretization of time can be obtained. The new time-counter clicks only when the feature value jumps in the feature-space.[3] This makes it useful to think of such features as event-indicators which represent the actuality of certain conditions of interest. This interpretation gives us an idea of how to define features in such a way that the dynamics at the level of the new counter simplified.

2.3 Modules

So far we have realized that the new "state-space" (the feature-space) and "time" can be made discrete. However, the action-space may still be infinite. Uniform discretization which lacks a priori knowledge would again be impractical in most of the cases (especially when the action space is unbounded), so we would rather consider an idea motivated by a technique which is often applied in artificial intelligence (AI) to solve large search-problems efficiently. The method in question divides the problem into smaller subproblems which in turn is divided into even smaller sub-problems, etc., then at the end rou-

[3]For continous-time systems some additional care is needed since arbitrarily fast transitions cannot be observed in practice. This places restrictions on the dynamics of the system and its features, but we do not concern ourselves with such structural questions here. In digital control time is already discretized, but the introduction of feature-jump based clocks is still worth the effort since the complexity of the feature level "dynamics" might be a great deal simpler than the original one.

tines are provided that deal with the resulting mini-problems. The solution of the entire problem is then obtained by working backwards: At every moment the mini-routine corresponding to the actual state of the search problem is applied. To put it in another way, the problem-solver defines a set of sub-goals, sub-sub-subgoals, etc. in such a way that if one of the sub-goals is satisfied then the resolution of the main-goal will be easy to achieve. In control-tasks the same decomposition can usually be done with respect to the main control objective without any difficulty. The routines that resolve the very low-level subgoals should be provided by closed-loop controllers which achieve the given subgoal under the set conditions (the conditions are usually given implicitly, e.g. the condition is fulfilled if the "predecessor" subgoals have already been achieved). The process of breaking up the problem into small subtasks can be repeated several times before the actual controllers are designed, so that the complexity of the individual controllers can be kept low. In complex problems it may happen that a particular controller proves to be useful in accomplishing several subtasks. In a mobile robot task a general-purpose controller could be that which unsticks the robot when it becomes stuck. The controllers together with their operating conditions, which may serve as a basic set of features, will be called modules. In principle a consistent transfer of the AI-decomposition yields the result that the operating conditions of the situation are exclusive and cover every situation. However, such a solution would be very sensitive to perturbations and unmodelled dynamics. A more robust solution can be obtained by extending the range of operating conditions and may mean that more than one controller can be applied at the same time. This calls for the introduction of a mechanism (the switching function) which determines which controller has to be activated if there are more than one available. More specifically, in our case a switching function S maps feature-vectors (which are composed of the concatenation of the operating conditions of the individual modules and some possible additional features) to the index of the module which should be activated when the actual feature-vector under consideration is observed. Of course only those modules can be activated whose operating conditions are satisfied. The operation of the whole mechanism is then the following. A controller remains active until the switching function switches to another controller. Since the switching function depends on the observed feature-values the controllers will certainly remain active till a change in the feature-vector is observed. We further allow the controllers to manipulate the observation-process. In this the controllers may inhibit an observation from occurring and thus may hold up their activity for a while.

The goal of the design procedure is to set up the modules and additional features in such a way that there exists a switching controller $S : F \to \{1, 2, \ldots, n\}$ which for any given history results in a closed-loop behaviour which fulfills the "goal" of control in time. It can be extremely hard to prove even the *existence* of such a valid switching controller. One approach is to use a so-called *accessability decision problem* for this purpose. The accessability decision problem is defined in the following way. Its state space is F (the possible values of the composed feature-vector), its action space is $\{1, 2, \ldots, n\}$, the indices of the controllers, and its transition-sets are defined as: the states that are accessible from feature-state f when using action i which are those elements f' of F for which there exists a history compatible with the feature f, such that when assuming the given history and using the controller indexed by i the next observed feature different from f will be f'. This means that the utilized controllers should be designed such that their operating conditions should not remain satisfied forever[4]. However, this is quite a natural condition since when translated back to the level of design it just means that each subtask should be completed in a finite time.[5] There

[4]One exception may be a controller whose porpose is just to maintain the "goal-state".

[5]Again, the goal-state may be retained for an infinitely long period of time.

are two ways of meeting this finite-finishing-time requirement. Firstly, design each controller with special attention paid to the problem, or secondly employ special features such as operating conditions which once "activated" terminate in some finite time. Continuing with the definition of the accessibility decision problem, the set of admissible "actions" corresponding to a feature-vector f consists of the index of those modules whose operating conditions are satisfied in f. Assume that the ultimate goal of control is to reach a certain subset of F (for this goal-indicator features should be provided). Let us now set-up a reward function r as a function of the features such that $r(f) = 1$ only if f is in the goal-set, otherwise $r(f) = 0$. In the context of hybrid control Sastry noted the following: If there exists a controller for the above accessability decision problem where the cumulated worst-case reward is non-zero (i.e. during evaluations only worst-case transitions are considered [6]) then there exists a switching controller which can solve the original problem [20]. Such a controller will be called proper in the worst-case sense. The reverse of the above implication is not necessarily true and this makes the analysis somewhat limited. Besides this the exact accessability decision problem can be very difficult to construct – usually it is easier to construct a slightly vaguer one, which has additional transitions in it. Another problem is that the exact transitions may well already revoke the existence of a controller which is proper in the above worst-case sense.

A weaker condition to the above, which may seem somewhat artificial at a first glance, is that the total probability of reaching the goal for all possible *accessability-compatible* transition probabilities should equal one. The idea behind it is that under certain conditions, such as ergodicity, we may assume that transitions can be modelled probabilistically. A transition-probability matrix is called accessibility-compatible if the transition-probability associated with a given transition (f, i, f') is non-zero if and only if f' is accessible from f using i in the sense of the definition given in the previous paragraph.

Clearly if there is a path in the graph underlying the accessibility transitions from each non-goal state f to one of the goal states, then there exists a switching policy under which the probability of reaching the goal states is one for each accessability-compatible transition-probability matrix. Such a switching strategy will be called an *almost surely proper* switching. Notice that due to the finiteness of the problem the expected number of steps to reach the goal will almost certainly be finite under any proper switching. If we knew that transitions take bounded physical time (this is not immediately obvious since the same transition may happen under an infinite number of different conditions because the process has an infinite state-space and also transitions can depend on the history of the process – another infinite set) then the expected total physical time will also be bounded.

Of course, since the definitions of the modules and features depend on the designer, it is reasonable to assume that by clever design a satisfactory decomposition and controllers could be found even if only qualitative properties of the controlled object were known. RL could then be used for two purposes: either to find the best switching function assuming that at least two proper switching functions exist, or to decide empirically whether a valid switching controller exists at all. The first kind of application of RL arises as result of the desire to guarantee the existence of a proper switching function through the introduction of more modules and features than is minimally needed. But then good switching which exploits the capabilities of all the available modules could well become too complicated to find manually.

If the accessability decision problem were truly extendible with transition-probabilities then RL could be rightly applied to find the best switching function. Unfortunately, although there is a theoretically well-founded way of introducing transition-probabilities when using a fixed (maybe stochastic) switching controller with some assumed noise, the resulting probabilities may well depend on the switching policy that prevent the convergence of the RL algorithms (see [21] for a discussion of these). However, the following "stability" theorem shows that the difference of the cost of optimal policies corresponding to different transition probabilities is proportional to the extent the transition probabilities differ, so we may expect that a slight change in the transition probabilities does not result in completely different optimal switching policies and hence, as will be explained shortly after the theorem, we may expect RL to work properly, after all.

Theorem 2.1 *Assume that two MDPs differ only in their transition-probability matrices, and let these two matrices be denoted by p_1 and p_2. Let the corresponding optimal cost-functions be v_1^* and v_2^*. Then*

$$||v_1^* - v_2^*|| \leq \gamma \frac{nC||p_1 - p_2||}{(1-\gamma)^2},$$

where $C = ||c||$ is the maximum of the immediate costs, $||\cdot||$ denotes the supremum-norm and n is the size of the state-space.

Proof:

Let T_i be the optimal-cost operator corresponding to the transition-probability matrix p_i, i.e.

$$(T_i v)(s) = \min_{a \in U(x)} \left(c(s,a) + \gamma \sum_{s' \in X} p_i(s,a,s') v(s') \right), \ v : \ S \rightarrow \Re, \ i = 1, 2.$$

Proceeding with standard fixpoint and contraction arguments (see e.g. [25]) we get that $\|v_1^* - v_2^*\| \leq \|T_1 v_1^* - T_1 v_2^*\| + \|T_1 v_2^* - T_2 v_2^*\|$ and since T_1 is a contraction with index γ, and the inequality $\|T_1 v - T_2 v\| \leq \gamma \|p_1 - p_2\| \sum_{y \in X} |v(y)|$ we obtain $\delta = \|v_1^* - v_2^*\| \leq \gamma\delta + \gamma\|p_1 - p_2\| |X| C/(1-\gamma)$, where $\|v_1^*\| \leq C/(1-\gamma)$

has been employed [19]. Rearranging the inequality in terms of δ then yields Theorem 2.1.

Q.E.D.

Motivated by the previous theorem we define ε-stationary MDPs as the quadruple (S, A, p, c), where S, A and c are as before but p, the transition probability matrix, may vary in time but with $||p_t - p^*|| \leq \varepsilon$ holding for all $t > 0$. Our expectations are that although the transitions cannot be modelled with a fixed transition probability matrix (i.e. stationary MDP), they can be modelled by an ε-stationary one even if the switching functions are arbitrarily varied.

By still assuming that a stationary MDP corresponds to a fixed switching we obtain, if a set switching policy is followed during learning, that the values learnt by RL will converge somewhere. However, on-line RL will change the exploration-policy continuously which may result in oscillating transition-probabilities. It is conjectured that the method developed by Szepesvári and Littman [11] can still be applied, but now with a reduced goal to show that if during learning the transition probabilities were oscillating slightly (say remaining within a set of diameter ε) then RL methods would result in oscillating estimates of the optimal value-function, but with the oscillation being asymptotically proportional to ε. Again this is an incomplete result as it leaves open the question of whether the oscillations in the transition probabilities are asymptotically small, which can be hard to answer since the actual policy executed during learning usually depends on the estimated values of the optimal cost-function, and the transition probabilities may depend on the learnt values, which all go to close the circle. A possible way out of this vicious circle is to make use of an assumption like, say, that the transition probabilities corresponding to different policies should not differ too much at all, i.e. to assume that the MDP is ε-stationary. This property

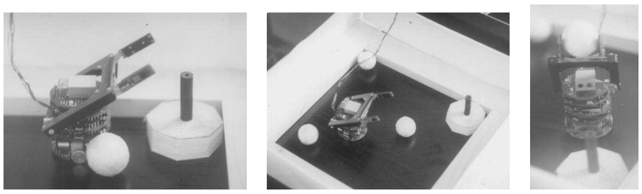

Figure 1: **The Khepera and the experimental environment.**

was clearly observed in our experiments which we will now describe.

3 Experiments

The validity of the proposed method was checked with actual experiments carried out using a Khepera-robot. The robot, the experimental setup, general specifications of the modules and the results are all presented in this section.

3.1 The Robot and its Environment

The mobile robot employed in the experiments is shown in Figure 1. It is a Khepera[6] robot equipped with eight IR-sensors, six in the front and two at the back, the IR-sensors measuring the proximity of objects in the range 0-5 cm. The robot has two wheels driven by two independent DC-motors and a gripper which has two degrees of freedom and is equipped with a resistivity sensor and an object-presence sensor. The vision turret is mounted on the top of the robot as shown. It is an image-sensor giving a linear-image of the horizontal view of the environment with a resolution of 64 pixels and 256 levels of grey. The horizontal viewing-angle is limited to about 36 degrees.

[6]The Khepera was designed and built at Laboratory of Microcomputing, Swiss Federal Institute of Technology, Lausanne, Switzerland.

This sensor is designed to detect objects in front of the robot situated at a distance spanning 5 to 50 cm. The image sensor has no tilt-angle, so the robot observes only those things whose height is exceeds 5 cm.

The learning task was defined as follows: find a ball in an arena, bring it to one of the corners marked by a stick and hit the stick with the ball. The robot's environment is shown in Figure 1. The size of the arena was 50 cm x 50 cm with a black coloured floor and white coloured walls. The stick was black and 7 cm long, while three white-coloured balls with diameter 3.5 cm were scattered about in the arena. The task can be argued to have been biologically inspired because it can be considered as the abstraction of certain foraging tasks or a "basketball play". The environment is highly chaotic because the balls move in an unpredictable manner and so the outcome of certain actions is not completely predictable, e.g. a grasped ball may easily slip out from the gripper.

3.2 The Modules

3.2.1 Subtask decomposition

Firstly, according to the principles laid down in Section 2, the task was decomposed into subtasks. The following subtasks were naturally: (T1) to find a ball, (T2) grasp it, (T3) bring it to the stick, and (T4) hit the stick with the grasped ball. Subtask (T3) was further broken into two subtasks, that of (T3.1) 'safe wandering' and (T3.2) 'go to the stick', since the robot cannot see the stick from every position and direction. Similarly, because of the robot's limited sensing capabilities, subtask (T1) was replaced by safe-wandering and subtask (T2) was refined to 'when an object nearby is sensed examine it and grasp it if it is a ball'. Notice that subtask 'safe wandering' is used for two purposes (to find a ball or the stick). The operating conditions of the corresponding controllers arose naturally as (T2) – an object should be nearby, (T3.2) – the stick should be detected, (T4) – the stick should be in front of the robot, and (T1,T3.1) – no condition. Since the behaviour of the robot must differ before and after locating a ball, an additional feature indicating when a ball was held was supplied. As the robot's gripper is equipped with an 'object-presence' sensor the 'the ball is held' feature was easy to implement. If there had not been such a sensor then this feature still could have been implemented as a switching-feature: the value of the feature would be `'on'` if the robot used the grasping behaviour and hence not the hitting behaviour. An 'unstuck' subtask and corresponding controller were also included since the robot sometimes got stuck. Of course yet another feature is included for the detection of "goal-states". The corresponding feature indicates when the stick was hit by the ball. This feature's value is `'on'` iff the gripper is half-closed but the object presence sensor does not give a signal. Because of the implementation of the grasping module (the gripper was closed only after the grasping module was executed) this implementation of the "stick has been hit by the ball" feature was satisfactory for our purposes, although sometimes the ball slipped out from the gripper in which case the feature turned 'on' even though the robot did not actually reach the goal. Fortunately this situation did not happen too often and thus did not affect learning.

3.2.2 Features and Controllers

The resulting list of modules and features, each of which we now elaborate upon, is shown in Table 2. The dynamics of the controller associated with `Module 1` was based on the maximization of a function which depended on the proximity of objects and the speed of both motors[7]. If there were no

[7]Modules are numbered by the identification number of their features.

obstacles near the robot this module made the robot go forward. This controller could thus serve as one for exploring the environment. `Module 2` was applicable only if the stick was in the viewing-angle of the robot, which could be detected in an unambiguous way because the only black thing that could get into the view of the robot was the stick. The range of allowed behaviour associated with this module was implemented as a proportional controller which drove the robot in such a way that the angle difference between the direction of motion and line of sight to the stick was reduced. The behaviour associated with `Module 3` was applicable only if there was an object next to the robot, which was defined as a function of the immediate values of IR-sensors. The associated behaviour was the following: the robot turned to a direction which brought it to point directly at the object, then the gripper was lowered. If the object-presence sensor attached to the gripper gave a signal then the robot judged that the object sensed was a ball and not the fence, so the robot closed the gripper and picked the object up. If the object-presence sensor did not signal the robot lifted up the gripper, then turned to a random direction and started going forward. The observation process was switched off until the whole procedure was finished. `Module 4` was the "hitting" module. Its feature function was `'on'` when the stick was near, i.e. the total activity of the linear-eye sensors exceeded a given constant[8], otherwise it was `'off'`. The associated behaviour was to let the gripper down and then raise it anew. This module was independent of whether there was a ball in the gripper or not, which entailed the robot having to learn that this module wasn't needed unless the ball was grasped. `Module` 5, as noted

[8]All the feature functions of the modules could be implemented as a threshold sum of the measurements of one sensor-modality. The thresholds were chosen in such a way that the task remained solvable, but no additional fine-tuning of these parameters was performed. Nevertheless, the values of these parameters could influence the resulting performance.

Feature No.	**Feature**	**Behaviour**
1	`‘on’` always	explore while avoiding obstacles
2	`‘on’` if the stick is in the viewing angle	go to the stick
3	`‘on’` if an object is near	examine the object grasp it if it is a ball
4	`‘on’` if the stick is near	hit the stick
5	`‘on’` if the robot is stuck	go backward
6	`‘on’` if the ball is grasped	-
7	`‘on’` if the stick is hit with the ball	-

Table 2: **Description of the features and the modules**.

earlier, was created to handle stuck situations. This module lets the robot going backward and is applicable if the robot has not been able to move the wheels into the desired position for a while. This condition is a typical time-window based feature. The sixth feature indicated the presence of the ball, which was `‘on’` if a ball was held, while `Feature 7` was the goal-detection feature described earlier.

The operating conditions of controllers were not exclusive; on the contrary there were many “states” when more than one behaviour was simultaneously applicable. On the other hand, some features were totally independent of each other. For example, if the robot was stuck then there must have been an object nearby, i.e. if `Feature 2` = `‘on’` then `Feature 4` = `‘on’` as well.

These dependencies mean that the "actual" state space was much smaller than 2^7 ($= 128$), the total number of possible states.

Simple case-analysis shows that there is no switching controller that would reach the goal with complete certainty (in the worst-case, the robot could return accidentally to state "10000000" from any state when the goal feature was `'off'`), so that an almost-sure switching strategy should always exist. On the other hand, it is clear that a switching strategy which eventually attains the target does indeed exist.

3.3 The Cost Structure

A dense cost-structure was applied: the cost of using each behaviour was one except when the goal was reached, whose cost was set to zero. Costs were discounted at a rate of $\gamma = 0.99$. Note that from time to time the robot by chance became stuck (the robot's 'stuck feature' was `'on'`), and the robot tried to execute a module which could not change the value of the feature-vector. This meant that the robot did not have a second option to try another module since by definition the robot could only make decisions if the feature-representation changed. As a result the robot could sometimes get stuck in a "perpetual" or so-called "jammed" state. To prevent this happening we built in an additional rule which was to stop and reinitialize the robot when it got stuck and could not unjam itself after 50 sensory measurements. A cost equivalent to the cost of never reaching the goal, i.e. a cost of $\frac{1}{1-\gamma}$ ($= 100$) was then communicated to the robot, which mimicked in effect that such actions virtually last forever.

3.4 Details of learning

Experiments were fully automated and organized in trials. Each trial run lasted until the robot reached the goal or the number of decisions exceeded 150 (a number that was determined experimentally), or until the robot became jammed. The 'stick was hit' event was registered by checking the state of the gripper (see also the description of `Feature 7`).

During learning the Boltzmann-exploration strategy was employed where the temperature was reduced by $T_{t+1} = 0.999\,T_t$ uniformly for all states [2].[9] During the experiments the cumulative number of successful trials were measured and compared to the total number of trials done so far, together with the average number of decisions made in a trial.

3.5 Results

Two sets of experiments were conducted. The first set was performed to check the validity of the module based approach, while the second was carried out to compare different RL algorithms. In the first set the starting exploration parameter T_0 was set to 100 and the experiment lasted for 100 trials. These values were chosen in such a way that the robot could learn a good switching policy, the results of these experiments being shown in Figure 2. One might conclude from the left subgraph which shows the percentage of task completions in different stages of learning that the robot could solve the task after 50 trials fairly well. Late fluctuations were attributable to unsuccessful ball searches: as the robot could not see the balls if they were far from it, the robot had to explore to find one and the exploration sometimes

[9] The theoretically-funded inverse logarithmic decrease was tried as well, but it was found to yield worse on-line performances than the faster geometric decrease which, in this particular specific case, seemed to guarantee the sufficiency of exploration.

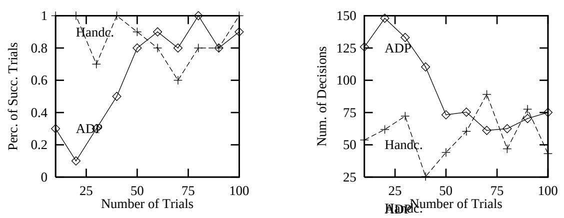

Figure 2: **Learning curves.** In the first graph the percentage of successful trials out of ten are shown as a function of the number of trials. In the second graph the number of decisions taken by the robot and averaged over ten trials are both shown, as well as a function of the number of learning trials.

took more than 150 decisions, yielding trials which were categorized as being failures. At the very beginning of learning the robot tried out the appropriate types of behaviour almost completely randomly, resulting in a large number of decisions per trial. The evaluation of behaviour-coordination is also observed in the second subgraph, which shows the number of decisions per trial as a function of time. The reason for later fluctuations is again due to a ticklish ball search. The performance of handcrafted switching policy is shown on the graphs as well. As can be seen the differences between the respective performances of the handcrafted and learnt switching functions are negligible. In order to get a more precise evaluation of the differences the average number of steps to reach the goal were computed for both switchings over 300 trials, together with their standard deviations. The averages were 46.61 and 48.37 for the learnt and the handcrafted switching functions respectively, with nearly equal std-s of 34.78 and 34.82, respectively.

One part of the learned policy is shown in Table 3, where 10 states were selected from the 25 explored ones together with their learnt associated behaviours. Theoretically, the total number of states is $2^7 = 128$, but as learning concentrates on feature-configurations that really occur this num-

No.	1	2	3	4	5	6	BId
1	1	0	1	0	0	0	3
2	1	1	0	0	0	0	2
3	1	1	1	1	1	0	3
4	1	0	1	0	1	0	5
5	1	1	0	1	0	0	2
6	1	0	1	0	0	1	1
7	1	1	0	0	0	1	2
8	1	1	1	1	1	1	4
9	1	0	1	0	1	1	5
10	1	1	0	1	0	1	4

Table 3: **One part of the learnt policy.** The six middle entries of the rows are the feature values corresponding to those features listed and numbered in Table 2. (The values of `Feature 7` are not shown as these are always zero here). The column marked by the label 'BId' denotes the number of the module that was chosen by a pure exploitation policy under the conditions described by the respective feature values. For example, in State 1 the robot used the behaviour "examine object" (Controller 3) with the pure exploitation strategy.

ber transpires to be just 25 here. The expected difference between the robot's behaviour before and after finding a ball can readily be seen. For example, in State 5 the robot moves towards the stick (compare Tables 2 and 3), because it has realized that this behaviour leads to an object with high confidence whose object usually happens to be a ball. In the dual state, State 10, which differs from State 5 only in that ball is now held, the robot properly chose the hitting action. Note that the learnt policy was always consistent with the handcrafted rules, but in certain cases the learnt rules are more refined than their handcrafted counterparts. The handcrafted rules are shown in the Table ?. For example, as the above example shows the robot learnt to exploit the fact that the arena was not completely level and as a result balls were biased towards the stick. The learnt actions in States 3 and 4 reveal another unexpected result: when the robot was stuck the presence of the stick made a difference. If the stick was in the viewing-angle the robot chose the object examination behaviour, otherwise it went backwards. This difference is again due to the fact that the balls are frequently situated around the stick and consequently the object-examination behaviour in a stuck state, and when the robot was close to the stick, would often lead to the gripping of a ball while simultaneously freeing the robot with high probability – so it was worth trying the object-examination behaviour in states like State 3.

In the rest of the experiments we compared two versions of ARTDP and three versions of *real-time Q-learning* (RTQL). The two variants of ARTDP were ADP, and ARTDP with the single case when $F_t := \{s_t\}$ and only one iteration in the inner cycle of the algorithm was performed. Note that due to the small number of states and module-based time discretization even ADP could be run in real-time. But variants of RTQL differ in the choice of the learning-rate's time-dependence. RTQL-1 refers to the choice of the so-called *search-then-converge* method, where $\alpha_k\left(s,a\right) = \frac{50}{100+n_k(s,a)}$, $n_k\left(s,a\right)$ being the number of times the event $(s,a) = (s_t, a_t)$ happened before time k plus one

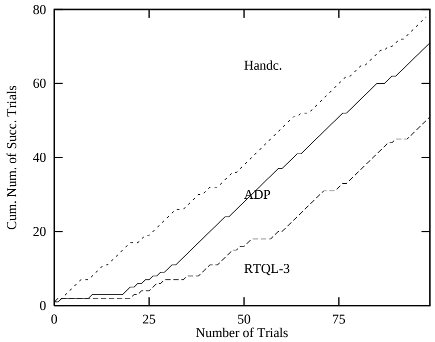

Figure 3: The cumulative number of successful trials in the case of learning with ADP and RTQL-2, and Boltzmann exploration with initial temperature $T_0 = 50$. The results for the handcrafted rule are also shown here.

(the parameters 50 and 100 were determined experimentally as being the best choices). In the other two cases (the corresponding algorithms were denoted by RTQL-2 and RTQL-3 respectively) constant learning rates (0.1 and 0.25, respectively) were utilized.

The online performances of the algorithms were measured as the cumulative number of successful trials. These values are depicted in Figure 3 during the learning procedure, and for the learning cases with ADP and RTQL-3. The starting exploration constant was set to $T_0 = 50$, a slope of $45°$ meaning that all of the trials were successful. Again, late drops in the graphs can be ascribed to unlucky ball searches. The bigger the curve slope the faster was the rate of learning, i.e. the smaller is the regret of learning. By definition the regret R_t at time t is the difference between the performance of an optimal agent (robot) and that of the learning agent accumulated up to trial t, i.e. it is the price of learning up to time t. If $s\,(t)$ denotes the number of successful trials out of the first t trials and the robot learns to behave "optimally" after trial number n_0 (i.e., it is able to hit the stick with the ball in every trial after time t_0) then $R = R_t = t - s(t) = t_0 - s\,(t_0)$ is the total regret (assuming that the optimal agent can reach the goal in every trial). A comparison of

Method	$T_0 = 100$	$T_0 = 50$	$T_0 = 25$	$T_0 = 0$
ADP	43	*23*	50	25
ARTDP	36	29	50	24
RTQL-1	53	69	47	66
RTQL-2	71	65	63	73
RTQL-3	62	49	*44*	51

Table 4: **Regret.** The table shows the number of unsuccessful trials among the first 100 trials.

the regrets for the different algorithms with different exploration ratios is given in Table 4. All algorithms were examined with all the four different exploration parameters ($T_0 = 100$, $T_0 = 50$, $T_0 = 25$, $T_0 = 0$) since the same exploration rate may well result in different regrets for different algorithms, as was also confirmed in the experiments.

Table 4 shows that the best performance was achieved with ADP and temperature $T_0 = 50$. This result (i.e. that ADP was the most successful) is not quite suprising because ADP makes the most intensive use of the robot's experiences out of the three distinct algorithm types. However, at the moment we do not have an explanation for why the moderated exploration approach worked best of all here. The difference between the performances with the other three exploration parameters seemed to be significant and rather intriguingly the medium exploration-ratio was found to be the best for the ARTDP case as well. More experiments would be needed to decide if the differences are statistically significant but running these experiments would require an enormous amount of time (approximately 40 days) and have not been performed yet. RTQL-3 yielded the best performance among the three RTQL algorithms, and this performance was obtained in the case of the smallest exploration-ratio ($T_0 = 25$). One explanation of the smaller optimal exploration-rate could be that the asymptotic dispersion of the learnt Q-values was proportional to the eventual learning rate, which was a large-valued constant in this case and big variances could have been the result of a kind of implicit exploration. The performance of the best Q-learner was easily overtaken by even the ARTDP algorithm, whose computational requirements were comparable to that of Q-learning. We have also tested another exploration strategy which Thrun found the best among several undirected methods[10] [27]. These runs reinforced our previous findings that estimating a model (i.e. running ADP or ARTDP instead of Q-learning) could reduce the regret rate by as much as 50%.

4 Related Work

There are two main research-tracks that influenced our work. The first was the introduction of features in RL. Learning while using features were studied by Tsitsiklis and Van Roy to deal with large finite state spaces, and also to deal with infinite state spaces [29]. Working on the output of features can well make the problem partially observable, so one should not expect that RL algorithms wich involve optimization will work in general (the theoretical results of Tsitsiklis and Van Roy concern only estimation types of algorithms, such as TD(λ), or non-adaptive algorithms [28]). Issues of learning in partially observable environments have been discussed by Singh et al. [21].

The second track is related to the use of local controllers together with a switching function which selects the controller to be activated at any arbitrary time. In connection with this topic, very recently and independently

[10] An exploration strategy is called undirected when the exploration does not depend on the number of visits to the state-action pairs.

of us, Sastry proposed the use of a hybrid-control approach to solve complex problems like highway-traffic control [20]. He proposed the design of several controllers, such as a car-following controller and an overtaking-controller, among others, which are imagined to work under different and well-specified conditions. He assumed that the system-dynamics was known and so the main concern of his approach was to find a switching controller which switched between the different controllers and which met certain requirements such as safety, maximal comfort (of the passangers) and maximal throughput (of the highway), the criteria importance having been ordered in this way. He used analytical tools to derive the accessability decision problem and suggested worst-case analysis to prove feasibility. Clearly, in contrast with his we assume here only a qualitative knowledge of the system-dynamics, which nevertheless enables us to design the modules and perform a preliminary feasibility analysis. Further we let let the system itself find out a good switching strategy by adapting to the actual environment.

A different approach was taken by Connel and Mahadevan whose work complements ours in that they set-up subtasks to be learned by RL and fixed the switching controller [17]. The main aim of their work was to prove that RL could be applied to learn good controllers at the noisy and unreliable sensor-actuator level, where also the state- and action-spaces were infinite. They found that statistics-based clustering combined with RL could solve this problem. In the future we plan to extend our work in this direction, i.e. only the subtask-decomposition and the features will be designed by hand, and RL should then be employed to learn both the low-level controllers and the switching controller possibly located simultaneously.

Asada et al. considered many aspects of mobile robot learning. They applied a vision-based state-estimation approach and defined "macro-actions" similar to our controllers [1]. In one of their papers they describe a goal-shooting problem in which a mobile robot shot a goal while avoiding another robot [5]. First, the robot learned two behaviours separately: the "shot" and "avoid" behaviours. Then the two behaviours were synthetised by a handcrafted rule and later this rule was refined via RL. The learnt action-values of the two behaviours were reused in the learning process while the combination of rules took place at the level of state variables.

Matarić considered a multi-robot learning task where each robot had the same set of behaviours and features [18]. Although the features of Matarić could clearly be interpreted as operating conditions of behaviours she did not restrict the applicability of the behaviours to the appropriate subspaces, which increased the complexity of the decision problem unnecessarily. Just as in our case, her goal was to learn a good switching function by RL. She considered the case when each of the robots learned separately and the ultimate goal was that learning should lead to a good collective behaviour, i.e. she concentrated mainly on the more involved multi-agent perspective of learning. She found that Q-learning worked badly compared to her "shaped reinforcement" approach which she found to be comparable to a handcrafted rule. In her approach time-averages of summed and appropriately defined immediate reinforcements served as the basis of decisions, i.e the handcrafted immediate reinforcements balanced the different aspects of the task and helped encode the structure of the switching function. This design required a lot a priori knowledge and experience, and seemed to be costly compared to the design of a good switching function. In her experiments with Q-learning Matarić used a sparse reward-structure, a unit reward was communicated to the learner when it reached the goal, otherwise a reward of zero was given. However it is well known that dense-rewards facilitate clever exploration rather than sparse-ones. (If dense rewards are used, the agent will be able to differentiate between tried and untried actions independently of whether the goal is reached or not, which can reduce the search-complexity in the trials considerably [10]). In contrast to her work, we followed a more engineer-like approach when we suggested designing the modules based on well-articulated and simple principles. Contrary to her findings we discovered that RL (with a dense-reward structure) can indeed work well at the modular level.

In the AI community there is an interesting approach to mobile robot control called Behaviour-Based Artificial Intelligence in which "competence" modules or behaviours have been proposed as the building blocks of "creatures" [15, 4]. Each of these modules has a list of preconditions similar to our own operating conditions cited here. The decision-making procedure is, on the other hand, usually quite different from ours. Maes for example proposed what she called a local-computation scheme. The modules were linked together through different channels and may inhibit or excite each other. Activation was spread along the channels and accumulated at the most relevant behaviour nodes. The behaviour whose activation first went above threshold was selected and executed. After finishing the behaviour the activation level of the behaviour was reset to zero and the whole process repeated. Like every *ad hoc* method, this method required careful tuning. Tyrrell found that in a complex decision task, aimed to simulate the task faced by a zebra living in the African Savannah, the decision making mechanism by Maes could not work well compared to other action-selection mechanisms. Moreover, he added that there might be theoretical reasons behind this failure [30]. Maes later pointed out that his findings could be debated on the basis of [14]. In an other work of hers she also proposed the learning of links between the modules [16], but to the best of our knowledge this algorithm has not yet been tried out on a real-robot.

The technique proposed here was also motivated by our earlier experiences with a value-estimation algorithm given in the form of "activation spreading" [26]. In this work activation spread out along the edges of a dynamically varying graph, where the nodes represented state transitions called triplets. Later the algorithm was extended so that useful subgoals could be found by learning [8, 9]. In the future we plan to extend the present module-based learning system with this kind of generalization capability. Such an extension may in turn allow the learning of a hierarchical organization of modules.

5 Summary and Conclusions

In this article module-based reinforcement learning was proposed to solve the coordination of multiple "behaviours" or controllers. The extended features served as the basis of time- and space discretization as well as the operating conditions of the modules. The construction principles of the modules were: decompose the problem into subtasks; for each subtask design controllers and the controllers' operating conditions; check if the problem could be solved by the controllers under the operating and observability conditions, add additional features or modules if necessary, set-up the reinforcement function and learn a switching from experience.

The idea behind our approach was that a partially observable decision problem could be usually transformed into a completely observable one if appropriate features (filters) and controllers were employed. Of course some *a priori* knowledge of the task and robot is always required to find those features and controllers. It was argued that RL could work well even if the resulting problem was only an almost stationary Markovian. The design principles were applied to a real-life robot learning problem and several RL-algorithms were compared in practice. We found that estimating the model and solving the optimality equation at each step (which could be done owing to the economic, feature-based time-discretization) yielded the best results. The robot learned the task after 700 decisions, which usually took less than

15 minutes in real-time. We conjecture that using a rough initial model good initial solutions could be computed off-line which could further decrease the time required to learn the optimal solution for the task.

The main difference between earlier works and our approach here is that we have established principles for the design modules and found that our subsequent design and simple RL worked spendidly. Plans for future research include extending the method via module learning and also the theoretical investigation of almost stationary Markovian decision problems.

Acknowledgment

The authors would like to thank Zoltán Gábor for his efforts of building the experimental environment. This work was supported by the CSEM Centre Suisse d'Electronique et de Microtechnique, Switzerland, OTKA Grants No. F20132 and No. 17110.

References

[1] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23:279–303, 1996.

[2] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 1(72):81–138, 1995.

[3] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[4] R. Brooks. Elephants don't play chess. In *Designing Autonomous Agents*. Bradford-MIT Press, 1991.

[5] M. A. E. Uchibe and K. Hosoda. Behavior coordination for a mobile robot using modular reinforcement learning. In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robot and Sytems*, pages 1329–1336, 1996.

[6] M. Heger. The loss from imperfect value functions in expectation-based and minimax-based tasks. *Machine Learning*, 22:197–225, 1996.

[7] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, November 1994.

[8] Z. Kalmár, C. Szepesvári, and A. Lőrincz. Generalization in an autonomous agent. In *Proc. of IEEE WCCI ICNN'94*, volume 3, pages 1815–1817, Orlando, Florida, June 1994. IEEE Inc.

[9] Z. Kalmár, C. Szepesvári, and A. Lőrincz. Generalized dynamic concept model as a route to construct adaptive autonomous agents. *Neural Network World*, 5:353–360, 1995.

[10] 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 15213, 12 1992. also appeared in "Machine Learning: A Special Issue on Reinforcement Learning, 1997 Vol 12. No. 3, pp. 234–345".

[11] M. Littman and C. Szepesvári. A Generalized Reinforcement Learning Model: Convergence and applications. In *Int. Conf. on Machine Learning*, 1996. http://iserv.iki.kfki.hu/asl-publs.html.

[12] M. L. Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, February 1996. Also Technical Report CS-96-09.

[13] M. L. Littman, A. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370, San Francisco, CA, 1995. Morgan Kaufmann.

[14] P. Maes. Adaptive action selection. In *Proc. of the Thirteenth Annual Conf. of the Cognitive Science Society*. Lawrence Erlbaum Associates, 1991.

[15] P. Maes. A bottom-up mechanism for behavior selection in an artificial creature. In J. Meyer and S. Wilson, editors, *Proc. of the First International Conference on Simulation of Adaptive Behavior*. MIT Press, 1991.

[16] P. Maes. Learning behavior networks from experience. In *Toward a Practice of Autonomous Systems (Proc. First European Conference on Artificial Life)*, pages 48–57. MIT Press, Cambridge, Massachusetts, 1992.

[17] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365, 1992.

[18] M. Matarić. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4, 1997.

[19] S. Ross. *Applied Probability Models with Optimization Applications.* Holden Day, San Francisco, California, 1970.

[20] S. Sastry. Algorithms for design of hybrid systems. In *Proc. of Int. Conf. of Information Sciences*, 1997.

[21] S. Singh, T. Jaakkola, and M. Jordan. Learning without state-estimation in partially observable markovian decision processes. In *Proc. of the Eleventh Machine Learning Conference*, pages pp. 284–292, 1995.

[22] S. Singh, T. Jaakkola, M. Littman, and Csaba Szepesvári. On the convergence of single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 1997. in preparation.

[23] R. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 1984.

[24] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8, 1996.

[25] C. Szepesvári and M. Littman. Generalized Markov Decision Processes: Dynamic programming and reinforcement learning algorithms. *Machine Learning*, 1997. in preparation.

[26] C. Szepesvári and A. Lőrincz. Behavior of an adaptive self-organizing autonomous agent working with cues and competing concepts. *Adaptive Behavior*, 2(2):131–160, 1994.

[27] S. Thrun. *The role of exploration in learning control*. Van Nostrand Rheinhold, Florence KY, 1992.

[28] J. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[29] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal difference learning with function approximation. Technical Report LIDS-P-2322, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1995.

[30] T. Tyrrell. *Computational Mechanisms for Action Selection*. PhD thesis, University of Edinburgh, 1993.

[31] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 3(8):279–292, 1992.

[32] P. Werbös. Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 22:25–38, 1977.