A Robot that Reinforcement-Learns to Identify and Memorize Important Previous Observations

Bram Bakker[1,2], Viktor Zhumatiy[1], Gabriel Gruener[3], Jürgen Schmidhuber[1]
[1] IDSIA, Manno-Lugano, Switzerland, {bram,viktor,juergen}@idsia.ch
[2] IAS, University of Amsterdam, The Netherlands
[3] CSEM Microrobotics, Alpnach, Switzerland, gabriel.gruener@csem.ch

Abstract— **It is difficult to apply traditional reinforcement learning algorithms to robots, due to problems with large and continuous domains, partial observability, and limited numbers of learning experiences. This paper deals with these problems by combining: 1. reinforcement learning with memory, implemented using an LSTM recurrent neural network whose inputs are discrete events extracted from raw inputs; 2. online exploration and offline policy learning. An experiment with a real robot demonstrates the methodology's feasibility.**

I. INTRODUCTION

Robot controllers are difficult to program. Reinforcement learning (RL) [5] is a promising alternative for robot controller development. RL agents do not require a teacher who demonstrates the correct action at each point of a trajectory (as in teacher-based supervised learning); instead, they just need scalar reward signals once they reach desired goals in an exploratory phase. Furthermore, many RL algorithms are explicitly designed to cope with noisy sensors and actuators and uncertain effects of actions.

There are significant discrepancies, however, between RL theory and the requirements of robot applications. First of all, most RL theory is concerned with small, discrete tasks. In contrast, robot domains are normally large and continuous. Secondly, most RL theory considers only tasks where the state of the environment is completely observable (Markov Decision Processes or MDPs). In most realistic robot applications, however, the state is only partially observable (POMDPs). For example, an office robot's sensors may tell it that it is in a corridor, but not in which; a Robocup robot's camera may currently see the ball but not the opponent's goal. In general, POMDPs require some sort of short-term memory of past observations and actions to determine the current state. But how to learn which past observations to store and which to ignore? Despite its ubiquity and importance, partial observability is rarely studied in robot learning research. A third problem is that RL algorithms typically take many learning steps to converge on good policies. Robot applications, however, allow for only a very limited number of learning experiences.

We deal with these problems by combining two main ideas:

1. Reinforcement learning with memory, based on extracted discrete events. Our RL algorithm uses a recurrent neural network which learns to detect relevant information from the past and store it in its recurrent activations. To facilitate this process, only significant changes in the robot's sensory inputs are extracted as discrete "events" and fed into the RL network. We investigated these algorithms before in simulation [1], [2], and here we transfer them to a real robot for the first time.

2. Online exploration/offline policy learning. RL algorithms usually require many learning iterations, but these iterations do not necessarily require real experiences. Therefore, we collect online data from a few real robot trials and use these to train the policy offline.

These two ideas are discussed more extensively in Sections 2 through 4. Section 5 describes an exemplary, partially observable robot task that reduces the problems described above to their essentials, and it describes our robot. Section 6 describes results, section 7 presents general conclusions.

II. EVENT EXTRACTION WITH ARAVQ

Robot sensory data typically consist of high-dimensional vectors sampled at a high rate. To make RL more feasible, it is desirable to obtain *spatially* and *temporally* compressed representations of these sensor vectors. In this way, the set of observations is reduced, and the sequence of observation-action pairs until goals are reached is compressed, facilitating temporal credit assignment. The method we use accomplishes both. It is called Adaptive Resource Allocation Vector Quantization, or ARAVQ [8]. It is an unsupervised learning method which classifies high-dimensional sensor vectors, for instance coming from a mobile robot's sensors, into a limited set of categories. The categories are associated with stored model vectors. A new model vector is dynamically allocated when a novel and stable situation is encountered. Furthermore, ARAVQ only passes on a new observation to the RL component when the sensor vector is stable and its classification changes. This is called an *event*.

ARAVQ maintains an input buffer of the last n input vectors. The vectors in the input buffer are averaged, yielding an input vector $\overline{x}(t)$ which already filters out some of the sensory noise. There is a set $M(t)$ of model vectors, which is initially empty. Additional model vectors are allocated when the following criteria are fulfilled. The input must be stable, i.e. $d_{\overline{x}(t)}$, the average Euclidian distance between $\overline{x}(t)$ and the last n inputs, must be below a threshold parameter ϵ. Secondly, the input must be novel, i.e. $d_{M(t)}$, the Euclidean distance between $\overline{x}(t)$ and the best matching stored model vector must be larger than $d_{\overline{x}(t)}$ plus a distance parameter δ. If both of these criteria are met, the filtered input is incorporated as an additional model vector:

$$M(t+1) = \begin{cases} M(t) \cup \overline{x}(t) & d_{\overline{x}(t)} \leq min(\epsilon, d_{M(t)} - \delta) \\ M(t) & \text{otherwise.} \end{cases} \tag{1}$$

When the current input is stable, a winning model vector $v(t)$ is selected, indicating the current classification category of the filtered input:

$$v(t) = \arg \min_{1 \leq j \leq |M(t)|} \{||\overline{x}(t) - m_j||\}; m_j \in M(t). \tag{2}$$

An event is detected when the winning model vector changes, and at that point a new observation is passed on to the RL component.

If the winning model vector matches the filtered input very closely, the filtered input is considered to represent a "typical" instance of the category, and the model vector is modified to become even more like this input:

$$\Delta m_{v(t)} = \begin{cases} \beta[\overline{x}(t) - m_{v(t)}] & ||\overline{x}(t) - m_{v(t)}|| < \frac{\epsilon}{2} \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

where β is a learning rate parameter.

III. REINFORCEMENT LEARNING WITH LSTM

The events extracted by ARAVQ feed into the RL system that learns the policy. Most realistic robot tasks are POMDPs, and therefore the policy must use short-term memory. ARAVQ provides a temporally compressed sequence of observations. Thanks to this temporal compression, it becomes easier for the RL component to detect temporal regularities in this sequence and learn what information from the past should be stored in short-term memory to disambiguate ambiguous states.

We use a Long Short-Term Memory (LSTM) recurrent neural network [4] as the RL component (see figure 1). LSTM has proven to be a powerful method for learning what information from sequential data to store in its internal memory and use later at the appropriate times. It has been successfully used in both supervised learning tasks [4] and reinforcement learning tasks [1], [2]. However, this is the first time LSTM is used as the controller for a real robot.

LSTM's short-term memory consists of *Constant Error Carrousels* (CECs), which are linear units whose activation does not decay autonomously over time. CECs receive inputs from the network's input units as well as from other units. In this way the CECs can store information for long periods of time. However, to prevent the CECs from filling up with useless information from the sequence of inputs, access to them is regulated using multiplicative *input gates*. The input gates *learn* to open and close at appropriate moments, such that useful information is stored in the CECs and useless information is discarded. Similarly, *output gates* learn to open and close at appropriate moments such that information stored in the CECs becomes available for the output units in the network. In this manner, the network's output can be a function of information stored many time steps ago. Finally, there are *forget gates* which can learn to reset the CECs when stored information is no longer useful. CECs together with their gates are called *memory cells* (see figure 2).

All these different types of units learn using an efficient version of Real-Time Recurrent Learning, i.e. a variation of the backpropagation algorithm for recurrent neural networks [4]. Thus, each weight is updated based on the estimated gradient of the error of the network's output units with respect to this weight. For example, the weight of a connection to an input gate may be increased because the estimated gradient of the error with respect to this weight says "if this weight is increased, the error will go down", and because of this, the input gate may learn to open for a particular input.

LSTM is one way of adding short-term memory to RL. Alternative methods include other, traditional recurrent neural network architectures [7], memory bits which the RL policy can learn to switch on and off [9], and U-tree [10], which builds a history tree of variable depth. Few if any of these learning techniques have been applied to real robots. In contrast to memory bits and U-tree, LSTM explicitly computes gradients with respect to past events, allowing more effective credit assignment, especially with respect to inputs from long ago and with many intermediate irrelevant inputs [1]. In contrast to traditional recurrent neural network architectures, which also explicitly compute gradients, the linear nature of CECs prevents computed gradients with respect to past events from decaying [4], [1], while the gates prevent disturbing influences to and from the CECs. These arguments were backed up by an empirical comparison of these various approaches in a previous study [1].

The LSTM network's activations are computed as follows. The net input $net_i(t)$ of any unit i at time t is calculated by

$$net_i(t) = \sum_m w_{im} y_m(t-1) \tag{4}$$

where w_{im} is the weight of the connection from unit m to unit i. A standard hidden unit's activation y_h, input gate activation y_{in}, output gate activation y_{out}, and forget gate

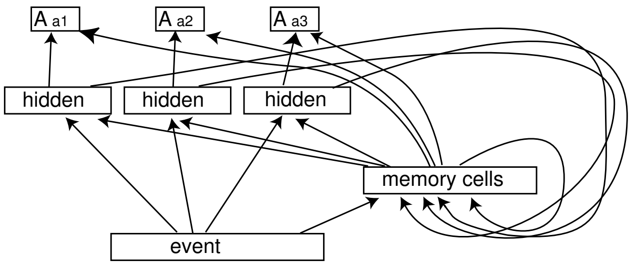

Fig. 1. Schematic architecture of the RL-LSTM network used in this paper. Each output unit represents the Advantage value of one action. Arrows indicate unidirectional, fully connected weights.

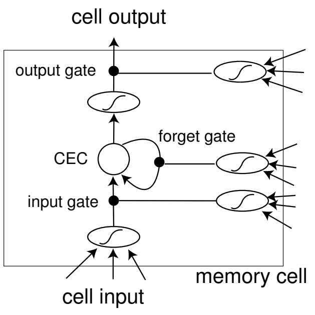

Fig. 2. One memory cell. It contains a linear CEC shielded from the input and output side of the network by gates.

activation y_φ is computed as

$$y_i(t) = f(net_i(t)) \tag{5}$$

where f is the standard logistic sigmoid function. The CEC activation $z_{c_j^v}$ is computed as follows:

$$z_{c_j^v}(t) = y_{\varphi_j}(t) z_{c_j^v}(t-1) + y_{in_j}(t) g(net_{c_j^v m}(t)) \tag{6}$$

where g is a logistic sigmoid function scaled to $[-2, 2]$. The memory cell's output $y_{c_j^v}$ is calculated by

$$y_{c_j^v}(t) = y_{out_j}(t) h(z_{c_j^v}(t)) \tag{7}$$

where h is a logistic sigmoid function scaled to $[-1, 1]$. An output unit's activation y_k, finally, is computed using

$$y_k(t) = net_k(t). \tag{8}$$

In an RL application (RL-LSTM), the LSTM network is used as the function approximator for a value function-based RL algorithm. We use Advantage learning [3], [1], which is a variation of the most widely used value function-based RL algorithm, Q-learning [5]. At each point in time, Advantage learning computes the following *temporal difference error*, which is the training error for the function approximator:

$$E^{TD}(t) = \frac{r(t+1) + \gamma V(s(t+1)) - V(s(t))}{\kappa} + V(s(t)) - A(s(t), a(t)) \tag{9}$$

where $A(s, a)$ is the Advantage value of action a in state s, $V(s) = \max_a A(s, a)$, and r is the immediate reward. γ is a discount factor in the range $[0, 1]$, and κ is a parameter scaling the difference between values of optimal and suboptimal actions. If $\kappa = 1$, (9) reduces to Q-learning.

As described above, the state s is approximated in LSTM using the current input (the current event provided by ARAVQ to the LSTM network) together with the activations of the memory cells, which encode information from past inputs. Each output unit k of the LSTM network represents the Advantage value of one action: $y_k(t) = A(s(t), a_k)$. We use eligibility traces, which have been shown to be beneficial in many RL tasks, especially in POMDPs [5], [1]. Each weight update of Advantage learning with eligibility traces and LSTM as the function approximator then becomes

$$w_{im}(t+1) = w_{im}(t) + \alpha E^{TD}(t) e_{im}(t) \tag{10}$$

for all weights w_{im}, where eligibility trace $e_{im}(t)$ is computed as

$$e_{im}(t) = \gamma\lambda e_{im}(t-1) + \frac{\partial y_K(t)}{\partial w_{im}} \tag{11}$$

where K is the index of the output unit representing the Advantage value of the current action. The exact computation of the partials $\frac{\partial y_K(t)}{\partial w_{im}}$, which can be derived straightforwardly from (4) through (8), is described in [4].

At each time step, the action with the highest Advantage value is chosen. In this paper, a chosen action selects one of three possible handcoded "behaviors" for execution until the next event detected by ARAVQ. However, a behavior can only actually be executed when it is "possible", i.e. when its own execution conditions are fulfilled. For example, the "turn left" behavior is only executed when the sensors say that there is open space to the left. Otherwise, the action with the second highest Advantage value is chosen, etc. This extra requirement, unusual in RL, was introduced to reduce the number of obviously senseless or possibly harmful actions and the amount of required online exploration (as described below).

IV. ONLINE EXPLORATION AND OFFLINE POLICY LEARNING

Most RL algorithms require many learning iterations, especially when combined, as done here, with gradient descent-based function approximators. In large part, this is because in these algorithms each learning step must be small, so as not to jump over the hill the algorithm is climbing. This suggests that not all learning iterations have to be based on new, real experiences; we may reuse old ones. Thus, a good approach to robot learning may be to obtain and store a limited number of real experiences by letting the robot explore its environment: online exploration. Next, the policy is learned offline, based directly on these stored experiences [6], [7] or on a more sophisticated model that was learned using these experiences [12]. In this paper, we use the stored experiences directly, i.e. the stored sequence of observation-action pairs are used by the RL algorithms as if they happen now: "experience replay" [6], [7]. Finally, the learned policy is transferred to the robot.

In some cases, there will be imperfections in the learned policy that come to light once it is transferred to the robot, imperfections due to the fact that only a small sample of real experiences is used for learning the policy. We may then finetune the policy through online RL on the robot, using essentially the same learning algorithm as the one used to learn the policy offline. This online RL may then be feasible because at this point we already have an almost correct policy.

When we use this methodology, RL algorithms must be applied somewhat differently. First of all, we must make sure that in the online exploration phase we collect a reasonable number of experiences, which are somewhat characteristic for the task. That is, regions in the state space that are essential for a good policy must be visited, and possible sensor and actuator noise must be reflected in the collected experiences.

During the offline policy learning phase, the most significant change is that the RL algorithm cannot explore in the standard way. Normally, the algorithm explores by taking the action with the highest state-action value with high probability, but taking another action with some small probability. In our case, exploration is done *implicitly*: sometimes the action taken by the robot at this point in the data set of collected experiences is different from the action that has the highest currently estimated state-action value ("exploration"), and sometimes it is equal ("exploitation"). In this way, the RL algorithm can still "explore", in some sense and to some degree, and estimate the values of different actions in a given state.

V. PARTIALLY OBSERVABLE T-MAZE

Our task reduces the above mentioned problems for RL robots to their essentials. A mobile robot must learn to navigate from a starting position to goal positions in a T-maze (see figures 3 and 4). It requires the robot to identify and memorize a particular bit of information, but prior to learning it is unclear which.

We use a Robocup-compliant wheeled CSEM robot called "Pele" with a differential drive. It is equipped with 4 Sharp GP2D02 infrared sensors, 2 on each side, providing a noisy measure of distance to the wall, sampled at around 5 Hz. In this way, the sensors provide the means to distinguish corridors from the T-junction. The robot also has a Texas Instruments TSL213 linear camera, whose

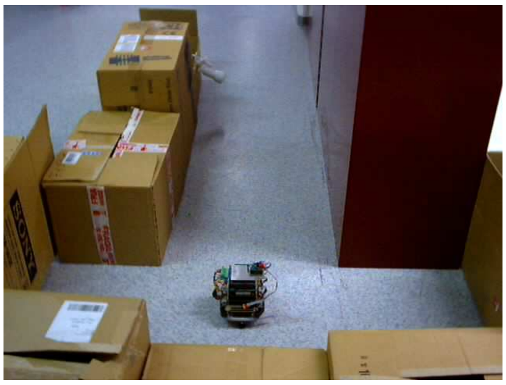
Fig. 3. The T-maze. The CSEM Pele robot is at the T-junction.

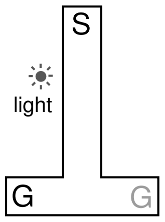

Fig. 4. Schematic representation of the T-maze. The robot must move from the starting position S to goal position G (the "wrong" goal position is shown in gray).

output was reduced to a scalar value which indicated the current maximum light intensity detected. There are 3 handcoded behaviors corresponding to the actions of the RL-LSTM controller: move forward, turn left 90°, and turn right 90°.

There are two equally probably goal positions at opposite sides of the T-junction. If the correct goal position is reached, the robot receives a reward of 1, if it ends up at the wrong goal position it receives a reward of -.5. At either point the episode ends, the robot's internal state is reset, and the robot is transferred to the starting position. There are no other rewards. The goal position cannot be perceived. However, in the corridor leading up to the T-junction there is a light (which can be detected by the linear camera) which is *on* when the goal is on the right, and *off* when the goal is on the left. If the robot discovers this correlation and learns to identify and store this relevant bit of information in short-term memory (embodied by activations circulating in its recurrent network), it can use it at the T-junction, *when the light is no longer visible*, to make the correct decision.

Thus, even though this is a conceptually simple task, it embodies essential challenges for RL robots, challenges that will reappear in most if not all realistic tasks, challenges that are unsurmountable by traditional RL: the task involves a real robot in a partially observable environment, and it is possible to resolve the ambiguity introduced by partial observability by remembering information from the past—but this information is not given in advance but must be discovered by the learning algorithm. In its raw version, the task has very sparse rewards as well as long-term dependencies between the appearance of the information that must be remembered and the moment at which this information must be used, making temporal credit assignment very difficult.

Variations of this task were investigated extensively before in simulation (e.g. [11], [8], [1], [2]), effectively making it a benchmark problem. In our previous IROS paper [2], a very similar task was investigated, using the same combination of ARAVQ and RL-LSTM, but that was all done in simulation. Thus, we can view this experiment as a test in the "real world": we test the algorithms developed in simulation studies in a robot, making use of the additional idea of online exploration/offline policy learning.

VI. EXPERIMENT AND RESULTS

Twelve episodes were collected in which the robot was driven through the T-maze to both goal positions with a handcoded controller (which used the handcoded behaviors described above), either with the light on or the light off. Reward signals at the end of each episode (both positive and negative ones) were provided by the experimenter manually and stored together with the sensory data and executed actions. This limited experience of 12 episodes was sufficient to train the controller offline.

The following parameter values were used for the RL-LSTM component: $\alpha = .0001$, $\gamma = .95$, $\lambda = .8$, $\kappa = .2$. For the ARAVQ component, the following parameter values were used: $\delta = .3$, $\epsilon = .8$, $n = 10$, $\beta = .001$. With these parameters, ARAVQ found 8 different model vectors, corresponding to 8 different observations for RL-LSTM. Compared to the large variety in raw sensor-vectors, this amounts to a significant *spatial compression*. The average number of events per episode detected by ARAVQ is 10.3. Compared to the large number of sampled sensor vectors per episode, this amounts to significant *temporal compression*.

Figure 5 plots raw sensor inputs (4 infrared sensor distance measurements and 1 linear camera maximum light intensity measurement) together with ARAVQ's corresponding model vectors over time, in part of the corridor with the light on. The data are scaled to fit. The graph illustrates the spatial and temporal compression achieved by ARAVQ event extraction. In this experiment, the sensor vector was still relatively low-dimensional (5 dimensions). But because ARAVQ is based on vector quantization, the number of model vectors does not necessarily increase rapidly with increasing numbers of dimensions. Thus, we expect the system to scale relatively well with higher-dimensional sensor inputs. Figure 5 also illustrates how the ARAVQ component filters out much of the sensor noise for the RL-LSTM component. Compared to our earlier

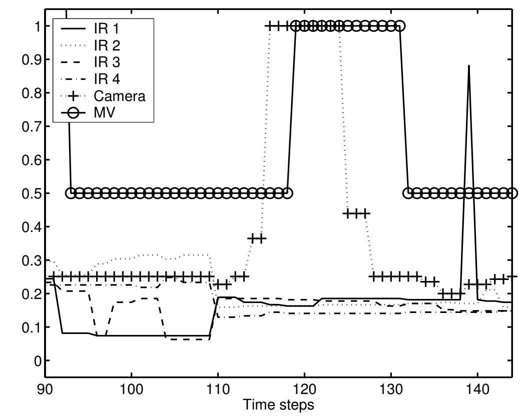

Fig. 5. Sensor inputs (infrared sensors IR, camera) over time in a part of the corridor when the light is on, together with extracted model vectors (MV). A change of model vector corresponds to an event.

simulation experiments [2], in which sensor noise was modeled as gaussian noise with small variance, in the real robot sensor noise has much larger variance and is much more irregular. Nevertheless, ARAVQ manages to extract a meaningful sequence of events, with occasional spurious events, in which RL-LSTM can discover and learn the temporal regularity between the light event and the best action at the T-junction.

Figure 6 plots the average cumulative reward per episode R, as a function of the number of learning iterations of offline RL-LSTM training, for one typical run. This measure is actually the estimated average cumulative reward *if the RL-LSTM network were in control of the robot*, which is not the case during offline policy learning in our methodology. Similar learning curves are found in different runs (of course converging at somewhat different moments in the run), and a coarse search in parameter space suggests a reasonable insensitivity to specific parameter settings.

Note how for most of the learning time R is simply the average of $r = 1$ and $r = -.5$, indicating random action selection at the T-junction. Then there is a rapid, almost discrete transition (but note the horizontal scale) to average cumulative reward $R = 1$, indicating an optimal solution, in which apparently LSTM has learned to remember the light information and use it correctly at the T-junction. Such rapid transitions in learning curves are typical for these types of learning systems. In many cases, solutions are discovered rather suddenly, as weights to the memory cells start to take on significant values and small correlations between CEC activations and rewards become visible for the gradient-based RL algorithm, suddenly boosting weight updates in the right directions.

The average cumulative reward converges to the optimal value $R = 1$ in around 200,000 iterations (20,000 episodes), taking approx. 20 seconds on a 2 GHz Pentium 4 PC. Note that it would be very difficult and time

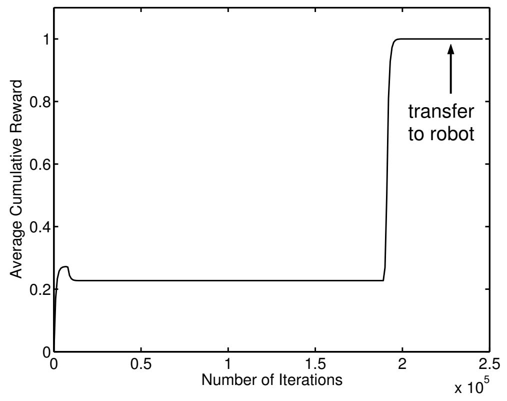

Fig. 6. Average cumulative reward per episode, as a function of iterations of the offline RL algorithm. After the learned controller is transferred to the robot (arrow), performance remains optimal.

consuming to accomplish this many learning steps on the real robot: a conservative estimate would be 8 days of non-stop online robot learning.

After convergence, the LSTM controller (trained offline) was transferred to the robot (indicated schematically in figure 6 by the arrow). In this simple task, no additional online training was necessary: the robot immediately had optimal performance. This shows the feasibility of the offline exploration/online policy learning methodology. The robot moves through the corridor, having learned to note and remember whether the light is on. Other information, such as the fact that the robot sees a corridor before encountering the junction is not relevant and therefore is not remembered. The robot arrives at the T-junction and if the light was on, it turns right and proceeds to the goal; if the light was off, the robot turns left and proceeds to the goal (see video).

Finally, some experimentation with the T-maze showed that the controller is fairly tolerant against changes to the exact starting position and the dimensions of the T-maze, without any additional training: e.g., the width of the corridors (see fig. 3) can be approx. 25% larger or smaller, and the walls do not have to be aligned perfectly. This robustness is partly due to the ARAVQ event extraction mechanism, which filters out much of the noise and irrelevant information for the RL-LSTM component, and partly due to the RL-LSTM controller itself, which has learned to ignore irrelevant events and only focus on remembering the light information and using that information at the T-junction.

VII. CONCLUSIONS

We combined memory-based reinforcement learning (RL) with online exploration and offline policy learning, to make RL feasible for robot domains. RL with memory was achieved through an LSTM recurrent neural network as a function approximator for Advantage learning, where the network received inputs from an ARAVQ event extraction mechanism. The system learned offline, based on a limited number of exploratory trials of the real robot in its environment. In our apparently simple but partially observable test domain the proposed methodology works: a task that would take days of online learning was offline-learned in seconds.

VIII. ACKNOWLEDGMENTS

This work was funded by a CSEM Microrobotics research grant.

IX. REFERENCES

[1] B. Bakker. Reinforcement learning with Long Short-Term Memory. In *NIPS 14*. 2002.

[2] B. Bakker, F. Linåker, and J. Schmidhuber. Reinforcement learning in partially observable mobile robot domains using unsupervised event extraction. In *Proc. IROS'02*, 2002.

[3] M. E. Harmon and L. C. Baird. Multi-player residual advantage learning with general function approximation. TR, Wright-Patterson Air Force Base, 1996.

[4] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9 (8):1735–1780, 1997.

[5] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

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

[7] L.-J. Lin and T. Mitchell. Reinforcement learning with hidden states. In *Proc. of the 2nd Int. Conf. on Simulation of Adaptive Behavior*. MIT Press, 1993.

[8] F. Linåker and H. Jacobsson. Mobile robot learning of delayed response tasks through event extraction: A solution to the road sign problem and beyond. In *Proc. IJCAI'2001*, 2001.

[9] M. L. Littman. An optimization-based categorization of reinforcement learning environments. In *Proc. of the 2nd Int. Conf. on Simulation of Adaptive Behavior*. MIT Press, 1993.

[10] R. A. McCallum. Learning to use selective attention and short-term memory in sequential tasks. In *Proc. 4th Int. Conf. on Simulation of Adaptive Behavior*, 1996.

[11] R. Rylatt and C. Czarnecki. Embedding connectionist autonomous agents in time: The road sign problem. *Neural Processing Letters*, 12:145–158, 2000.

[12] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc. ICML 7*, 1990.