Reinforcement Learning with LSTM in Non-Markovian Tasks with Long-Term Dependencies

Bram Bakker
Unit of Experimental and Theoretical Psychology
Leiden University
P.O. Box 9555; 2300 RB, Leiden; The Netherlands
bbakker@fsw.leidenuniv.nl

Abstract

This paper presents reinforcement learning with a Long Short-Term Memory recurrent neural network: RL-LSTM. Model-free RL-LSTM using Advantage(λ) learning and directed exploration can solve non-Markovian tasks with long-term dependencies between relevant events. This is demonstrated in a T-maze task, as well as in a difficult variation of the pole balancing task.

1 Introduction

1.1 Non-Markovian reinforcement learning tasks

Reinforcement learning (RL) is a way of learning how to behave based on delayed, scalar reward signals (see Sutton & Barto, 1998; Kaelbling, Littman, & Moore, 1996 for reviews). Among the more important challenges for RL are tasks where the state of the environment is only partially observable. In other words, part of the environmental state is *hidden* from the agent. Such tasks are called non-Markovian tasks or Partially Observable Markov Decision Processes (POMDPs).

Many real world tasks have this problem of hidden state. For instance, in a maze navigation task different positions in the maze may look the same but require different actions. The decision by a gift-wrapping agent whether or not to open an observed closed box should depend on whether or not the agent has already put the gift in the box (Lin & Mitchell, 1993). Thus, hidden state makes RL more realistic. However, it also makes it more difficult, because now the agent not only needs to learn the mapping from environmental states to actions, it also needs to determine at each point which environmental state it is in.

The "classical" approach[1] from operations research to solving this problem assumes the availability of an exact model of the environment, containing the probabilities of reward, state transitions, and observations, given the different states and actions. Given this model,

[1] Unfortunately, the term POMDPs is used in the literature both to indicate non-Markovian tasks and to indicate this particular approach to solving such tasks. This ambiguity is the reason for using the term "non-Markovian tasks" in this paper.

the "belief state" can be computed, which indicates the relative probabilities of being in different environmental states, and from which the optimal policy can be computed using techniques derived from dynamic programming (Lovejoy, 1991).

If no a priori model of the environment is available, there are different ways to proceed. One way is to build a model online, which learns to predict observations and rewards, and in this way learns to infer the environmental state at each point. A separate controller can then use these inferred environmental states as the basis for its control policy (Lin & Mitchell, 1993; Chrisman, 1992; Suematsu & Hayashi, 1999; Schmidhuber, 1991a, 1991b).

Another way to proceed is to abandon the idea of such a full-blown predictive model of the environment. In certain restricted cases, the non-Markovian task can be hierarchically decomposed into a set of Markovian subtasks, each of which can be solved by a reactive controller mapping observations to actions (Wiering & Schmidhuber, 1997). Another model-free approach is to attempt to resolve the hidden state by making the chosen action depend not only on the current observation, but also on some representation of the history of observations and actions. The general idea is that the current observation together with this representation of the history may yield a Markovian state signal. The most straightforward way to realize the idea is to use a fixed depth history window ("delay lines") of the most recent observations and actions (Littman, 1993; Lin & Mitchell, 1993). A more sophisticated variation of this is to use a *variable* depth history window (McCallum, 1996), which allows the history window's depth to be different in different parts of the environmental state space.

The representation of the history of observations and actions does not have to literally correspond to the most recent observations and actions. It may instead correspond to memory bits that the controller has learned to switch on and off (Littman, 1994; Lanzi, 2000; Cliff & Ross, 1994; Peshkin, Meuleau, & Kaelbling, 1999). Alternatively, it may correspond to recurrent activations from hidden units in recurrent neural networks (Lin & Mitchell, 1993; Bakker & van der Voort van der Kleij, 2000; Gomez & Miikkulainen, 1999). Finally, the policy may be implemented as a finite state automaton (FSA), where the state of the FSA represents the history of observations and actions (Meuleau, Peshkin, Kim, & Kaelbling, 1999; McCallum, 1993).

1.2 Long-term dependencies

Most of the approaches described above have problems if there are long-term dependencies between relevant events. An example of a long-term dependency problem is a maze navigation task where the only way to distinguish between two T-junctions that look identical is to remember an observation or action a long time before either T-junction.

In such a case, there is no straightforward way to decompose the non-Markovian task into Markovian subtasks using Wiering and Schmidhuber's (1997) approach: the agent simply must remember the relevant piece of information up until the T-junction. There are also obvious problems with fixed size history window approaches: if the relevant piece of information to be remembered falls outside the history window, the agent cannot use it. McCallum's (1996) variable history window has, in principle, the capacity to represent long-term dependencies. However, the system starts out with zero history and increases the depth of the history window step by step, based on gathering statistics. This process will make learning long time lag dependencies difficult when there are no short-term dependencies to build on, and even it works, it will be a very time consuming process. In addition, a history

window approach must represent the entire history from the relevant piece of information onwards, even if intermediate events are irrelevant, yielding an unnecessarily large policy representation.

Model-free approaches based on memory bits, recurrent neural networks, and FSAs do not have to represent (possibly long) entire histories, but can in principle extract and represent just the relevant information for an arbitrary amount of time. The same is true for a number of approaches that learn a predictive model of the environment. However, *learning* to to extract and represent information from a long time ago has proven difficult, both for model-based and for model-free approaches. The difficulty lies in discovering the correlation between a piece of information and the moment at which this information becomes relevant at a later time, given the distracting observations and actions between them.

This difficulty can be viewed as an instance of the general difficulty of learning long-term dependencies in timeseries data (Bengio, Simard, & Frasconi, 1994). This paper proposes the use of one particular solution from the neural networks literature that has worked well in *supervised* timeseries learning tasks: Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997; Gers, Schmidhuber, & Cummins, 2000). In this paper an LSTM recurrent neural network is used in conjunction with model-free RL, in the same spirit as the model-free recurrent neural network approaches described above (Lin & Mitchell, 1993; Bakker & van der Voort van der Kleij, 2000).

1.3 Outline

The next section describes LSTM. Section 3 presents LSTM's combination with reinforcement learning in a system called RL-LSTM. Section 4 contains simulation results on non-Markovian RL tasks with long-term dependencies. Section 5, finally, presents the general conclusions.

2 LSTM

2.1 Memory cells

LSTM is a recently proposed recurrent neural network architecture, originally designed for supervised timeseries learning (Hochreiter & Schmidhuber, 1997; Gers et al., 2000). It is based on an analysis of the problems that conventional recurrent neural network learning algorithms (e.g. backpropagation through time and real-time recurrent learning) have when learning timeseries with long-term dependencies. These problems boil down to the problem that errors propagated back in time tend to either vanish or blow up (see (Hochreiter & Schmidhuber, 1997)).

LSTM's solution to this problem is to enforce *constant* error flow in a number of specialized units, called Constant Error Carrousels (CECs). This actually corresponds to these CECs having linear activation functions which do not decay over time. In order to prevent the CECs from filling up with useless information from the timeseries, access to them is regulated using other specialized, multiplicative units, called input gates. Like the CECs, the input gates receive input from the timeseries and the other units in the network, and they *learn* to open and close access to the CECs at appropriate moments.

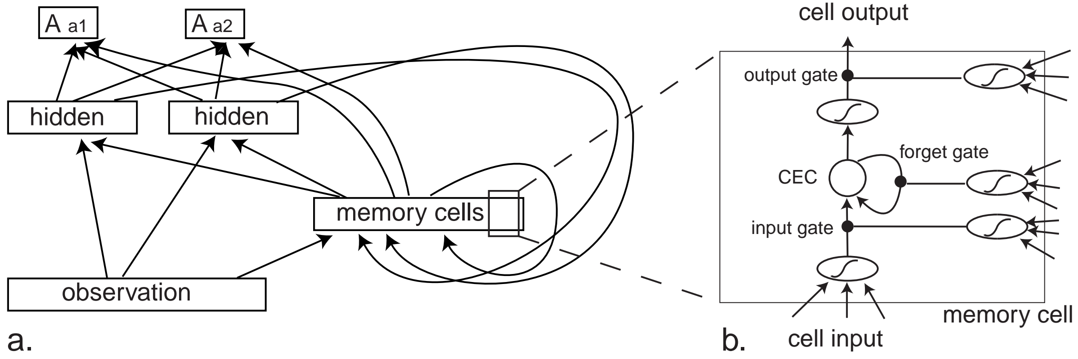

Figure 1: a. The general LSTM architecture used in this paper. Arrows indicate unidirectional, fully connected weights. The network's output units directly code for the Advantages of different actions. b. One memory cell.

Access *from* the activations of the CECs to the output units (and possibly other units) of the network is regulated using multiplicative output gates. Similar to the input gates, the output gates learn when the time is right to send the information stored in the CECs to the output side of the network. A recent addition is forget gates (Gers et al., 2000), which learn to reset the activation of the CECs (in a possibly gradual way) when the information stored in the CECs is no longer useful. The combination of a CEC with its associated input, output, and forget gate is called a memory cell. See figure 1b for a schematic of a memory cell. It is also possible for multiple CECs to be combined with only one input, output, and forget gate, in a so-called memory block.

2.2 Activation updates

Figure 1a shows the general network architecture used in this paper. The network's activations are computed as follows. The net input $net_i(t)$ of any unit i at time t is calculated by

$$net_i(t) = \sum_m w_{im} y^m(t-1) \tag{1}$$

where w_{im} is the weight of the connection from unit m to unit i.[2] A standard hidden unit's activation y^h, output unit activation y^k, input gate activation y^{in}, output gate activation y^{out}, and forget gate activation y^φ is computed as

$$y^i(t) = f_i(net_i(t)) \tag{2}$$

where f_i is the standard logistic sigmoid function, squashing the net input to the range $[0, 1]$. The standard hidden units receive input from the input layer and the memory cell outputs.

[2] For ease of notation, the current input layer activations are viewed as activations from one timestep ago, and likewise for inputs to the output units.

The gates receive the same input, plus input from the hidden layer and all the gates. The CEC activation $s_{c_j^v}$, or the "state" of memory cell v in memory block[3] j, is computed as follows:

$$s_{c_j^v}(t) = y^{\varphi_j}(t) s_{c_j^v}(t-1) + y^{in_j}(t) g(net_{c_j^v m}(t)) \tag{3}$$

where g is a logistic sigmoid function scaled to the range $[-2, 2]$, and $s_{c_j^v}(0) = 0$. Note how the memory cell's input gate activation y^{in_j} determines, in a multiplicative way, to what extent the net input "enters" the memory cell. This net input comes from the input layer, standard hidden layer, the gates, and the memory cell outputs. The memory cell's output $y^{c_j^v}$ is calculated by

$$y^{c_j^v}(t) = y^{out_j}(t) h(s_{c_j^v}(t)) \tag{4}$$

where h is a logistic sigmoid function scaled to the range $[-1, 1]$. Similar to the input gate, the output gate activation y^{out_j} determines, in a multiplicative way, to what extent the memory cell's contents are made available for other units, most notably the output units. An output unit's activation y^k, finally, is computed using

$$y^k(t) = f_k(net_k(t)). \tag{5}$$

In this paper, f_k is the identity function, so output units are linear. An output unit receives input from a dedicated layer of standard hidden units (for reasons explained in section 3.2), and from all the memory cell outputs.

2.3 Learning

At some or all timesteps of the timeseries, the output units of the network may make prediction errors. Errors are propagated just one step back in time through all units other than the CECs, including the gates. However, errors are backpropagated through the CECs for an indefinite amount of time, using a variation of real-time recurrent learning. Unlike standard backpropagation through time and real-time recurrent learning, the resulting learning algorithm for LSTM (Hochreiter & Schmidhuber, 1997; Gers et al., 2000) is local in both space and time, and its update complexity per weight and timestep is $O(1)$. The learning algorithm is adapted somewhat for reinforcement learning, as explained in the next section.

3 RL-LSTM

3.1 Model-free RL-LSTM

LSTM could be applied to RL tasks in various ways. One way would be to use LSTM as the basis of a model-based system. An LSTM network could learn a predictive model of the environment, and in this way learn to infer the environmental state at each point (similar

[3] Memory cells are not organized in memory blocks in this paper. Therefore, index j could be omitted. It is left in in order to maintain generality as well as compatibility with the standard LSTM formulation (see Hochreiter & Schmidhuber, 1997; Gers et al., 2000).

to Lin & Mitchell, 1993; Chrisman, 1992; Schmidhuber, 1991a). LSTM's architecture would allow the predictions of observations and rewards to depend on information from long ago. The system could then learn the mapping from (inferred) environmental states to actions using standard RL algorithms designed for Markovian tasks, such as Q-learning (Lin & Mitchell, 1993; Chrisman, 1992), or by backpropagating through the model to a separate controller (Schmidhuber, 1991a, 1991b).

An alternative, model-free approach—and the one used here—is to use LSTM to directly represent the policy. The policy then depends on the current observation, which is the input to the network, and the recurrent activations in the network, which represent the agent's history (as in Lin & Mitchell, 1993; Bakker & van der Voort van der Kleij, 2000). One possible advantage of such a model-free approach over a model-based approach is that the system may learn to only resolve hidden state insofar as that is *useful* for obtaining higher rewards, rather than waste time and resources in trying to predict features of the environment that are irrelevant for obtaining rewards (McCallum, 1996).

An interesting feature of recurrent neural networks, such as LSTM networks, is that they present a unified solution to two opposing problems in RL tasks: an environment may, *at the same time*, provide both too little and too much sensory information to the agent.[4] Too little sensory information corresponds to the problem of hidden state discussed extensively above. The recurrent processing of the network can provide the agent with the necessary representation of the past. Too much sensory information corresponds to the problem that the environment may have many features which can be sensed but which are irrelevant with respect to performance. It also corresponds to the related problem that most realistic environments have many states (especially if we are dealing with continuous environments), such that generalization is necessary. The feedforward processing of the network can provide the necessary generalization and "selective attention" to relevant features in the current observation. LSTM's main contribution, of course, is related to the problem of having too little sensory information, but as we shall see in section 4.2, where an example is presented with continuous environmental states, the ability to deal with too much sensory information is there as well.

3.2 Advantage learning

In Lin and Mitchell (1993) and Bakker and van der Voort van der Kleij (2000), Simple Recurrent Networks learned to approximate the Q-function of the well-known RL algorithm called Q-learning (Watkins, 1989). For the test problems presented here, a related algorithm is used, called Advantage learning (Harmon & Baird, 1996). Advantage learning is derived from Advantage updating, which in turn was designed as an improvement on Q-learning for continuous-time RL. In continuous-time RL, values of adjacent states typically differ by only small amounts, relative to the possible overall variance of values. This means that the optimal Q-values of different actions in a given state, from which the policy is directly derived, differ by only small amounts. These differences, then, can easily get lost in the noise, especially with function approximators such as neural networks. Advantage learning remedies this problem by artificially decreasing the values of suboptimal actions in each state.

[4] This argument follows McCallum (1996), whose algorithm shares this feature with recurrent neural networks.

The differences between the values of different actions in each state are thus greater than in Q-learning, and less likely to get lost in the noise. Doya (2000) shows that Advantage updating (so presumably Advantage learning as well) can be understood as one approach to deriving an efficient control policy for continuous-time systems using the estimated *gradient* of the value function.

Even though Advantage learning was originally designed for continuous-time RL, in this work it is used for both continuous-time and discrete-time RL. Note that the same problem of small differences between the values of adjacent states applies to any RL problem with long paths to rewards. And to demonstrate RL-LSTM's potential to bridge long time lag dependencies, we need to consider such RL problems. Furthermore, to the best of my knowledge, this is the first time Advantage learning is used in a non-Markovian task.

In Advantage learning, the value of an environmental state is defined as

$$V^*(s) = \max_a A^*(s,a). \tag{6}$$

The true Advantage $A^*(s,a)$ for each action a in state s is defined as

$$A^*(s,a) = V^*(s) + \frac{\langle r + \gamma V^*(s') \rangle - V^*(s)}{\kappa} \tag{7}$$

where $\langle \cdot \rangle$ represents the expected value over all possible results of executing action a in state s, which leads to immediate reward r and a new state s'. γ is a discount factor in the range $[0,1]$. For an optimal action, the second term is zero, meaning that the value of this action is the value of the state. For suboptimal actions, the second term is negative. The size of the second term then depends on κ, a constant scaling the difference between values of optimal and suboptimal actions (κ replaces $K\Delta t$ of the original formulation).

The LSTM network's output units directly code for the Advantages of different actions (see figure 1a). In non-Markovian RL, the agent does not have access to the complete environmental state s. Thus, s is approximated by the current observation, which is the input to the network, together with the representation of the agent's history, encoded in the recurrent activations of the network. Each output unit coding for the Advantage of a single action has its own dedicated standard hidden units layer. This is done because previous experience with combining Q-learning and neural networks has shown that if there is only a single hidden layer, conflicting error signals for different actions can make learning difficult (Lin, 1992; Abul, Polat, & Alhaji, 2000).[5]

To implement Advantage learning in the LSTM network, the righthand side of equation 7 can be viewed as the target output for the output unit coding for the Advantage of the executed action at the current timestep. As in Q-learning, actual experiences in the environment replace $\langle \dots \rangle$, and the system's own current approximation to $V^*(s)$ and $V^*(s')$ is used. The other output units do not receive error signals. Thus, the update is based on the immediate reward received after executing action a, and the maximally attainable Advantage value in the next state, estimated by the network itself. This yields the *temporal difference error* $T(t)$, the difference between the estimated righthand side and lefthand side of equation

[5]Another approach to this problem is to give an action representation as *input* to the network (Bakker & van der Voort van der Kleij, 2000).

7:[6]

$$T(t) = V(s(t)) + \frac{r(t) + \gamma V(s(t+1)) - V(s)}{\kappa} - A(s(t), a(t)) \tag{8}$$

3.3 Eligibility traces and the learning algorithm

In this work, Advantage learning is also combined with *eligibility traces*, which are often used to speed up learning in RL (Sutton & Barto, 1998; Watkins, 1989). The resulting algorithm is called Advantage(λ) learning. Similar to Q(λ)-learning, Advantage(λ) learning with a function approximator requires the storage of one eligibility trace e_{im} per weight w_{im}. In gradient descent learning, the standard approach to optimizing weights in neural networks and also in LSTM, a weight update corresponds to

$$w_{im}(t+1) = w_{im}(t) + \alpha \frac{\partial E(t)}{\partial y^k(t)} \frac{\partial y^k(t)}{\partial w_{im}} \tag{9}$$

where $E(t)$ is the sum of squared errors and α is a learning rate parameter. If the Advantage A^k of action k is a linear function of output y^k, then

$$\frac{\partial E(t)}{\partial y^K(t)} = CT(t) \tag{10}$$

where K represents the chosen action and C is a constant which can be absorbed in the learning rate parameter. As explained in section 3.2, $\frac{\partial E(t)}{\partial y^k(t)} = 0$ for all $k \neq K$. With eligibility traces, a weight update becomes

$$w_{im}(t+1) = w_{im}(t) + \alpha T(t) e_{im}(t) \tag{11}$$

where

$$e_{im}(t) = \gamma \lambda e_{im}(t-1) + \frac{\partial y^K(t)}{\partial w_{im}} \tag{12}$$

in which λ is a parameter determining how fast the eligibility trace decays (Sutton, 1989; Sutton & Barto, 1998). $e_{im}(0) = 0$, and $e_{im}(t-1)$ is set to 0 if an exploratory action is taken (Sutton & Barto, 1998; Watkins, 1989). If $\lambda = 0$, the first term disappears and we are left with LSTM's normal gradient descent update.

Equation 12 basically says that an eligibility trace keeps a decaying record of how a weight w_{im} has influenced the recent outputs of the function approximator, and thus, to what extent it is eligible for change when a temporal difference error—e.g. caused by an unexpected reward—comes in. LSTM's learning algorithm truncates $\frac{\partial y^k(t)}{\partial w_{im}}$ at all points in the network other than the CECs (see section 2.3). Since the same approximations to $\frac{\partial y^k(t)}{\partial w_{im}}$ are used in equation 12, this means that eligibility traces are truncated at the same points.

[6] Certain RL algorithms based on temporal difference errors, such as Q-learning and Advantage learning, can in principle lead to divergence when used in conjunction with function approximators such as neural networks (e.g. see Harmon & Baird, 1996; Sutton & Barto, 1998). In the studies reported here, this did not occur. If it does occur, it is fairly straightforward to instead use a safer, "residual" version of Advantage learning (Harmon & Baird, 1996).

In addition to modifying LSTM's learning algorithm in this way, eligibility traces require a minor rearrangement of the variables and the computation, when compared with the original LSTM formulation. This is so because in contrast with the original LSTM formulation, the partial derivatives $\frac{\partial y^K(t)}{\partial w_{im}}$ must now be calculated separately and explicitly. None of these changes affect LSTM's update complexity.

For the network architecture depicted in figure 1a and described in section 2.2, the partial derivatives $\frac{\partial y^K(t)}{\partial w_{im}}$ are computed as follows. For the weights from memory cell outputs and from the dedicated layer of standard hidden units to output units ($i = K$),

$$\frac{\partial y^K(t)}{\partial w_{Km}} = f'_K(net_K(t))y^m(t-1). \tag{13}$$

For the weights from input units and memory cell outputs to standard hidden units h,

$$\frac{\partial y^K(t)}{\partial w_{hm}} = w_{Kh}f'_K(net_K(t))f'_h(net_h(t))y^m(t-1). \tag{14}$$

For the weights from input units, memory cell outputs, gates, and standard hidden units to output gate units out_j,

$$\begin{aligned}\frac{\partial y^K(t)}{\partial w_{out_j m}} = &\sum_{v=1}^{S_j} h(s_{c_j^v}(t))\Big(w_{Kc_j^v}f'_K(net_K(t)) + \sum_h w_{Kh}f'_K(net_K(t))w_{hc_j^v}f'_h(net_h)(t)\Big) \\ &\cdot f'_{out_j}(net_{out_j}(t))y^m(t-1)\end{aligned} \tag{15}$$

where S_j is the number of CECs (1, in this paper) in memory block j. For the weights from input units, memory cell outputs, gates, and standard hidden units to CEC units c_j^v,

$$\begin{aligned}\frac{\partial y^K(t)}{\partial w_{c_j^v m}} = &\Big(w_{Kc_j^v}f'_K(net_K(t)) + \sum_h w_{Kh}f'_K(net_K(t))w_{hc_j^v}f'_h(net_h)(t)\Big) \\ &\cdot y_{out_j}(t)h'(s_{c_j^v}(t))\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}}\end{aligned} \tag{16}$$

where $\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}}$ is the information that needs to be stored in LSTM's version of real time recurrent learning used within the memory cells. It is updated as follows:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}}y^{\varphi_j}(t) + g'(net_{c_j^v}(t))y^{in_j}(t)y^m(t-1). \tag{17}$$

For the weights from input units, memory cell outputs, gates, and standard hidden units to input gates in_j,

$$\begin{aligned}\frac{\partial y^K(t)}{\partial w_{in_j m}} = &\sum_{v=1}^{S_j}\Big(w_{Kc_j^v}f'_K(net_K(t)) + \sum_h w_{Kh}f'_K(net_K(t))w_{hc_j^v}f'_h(net_h)(t)\Big) \\ &\cdot y_{out_j}(t)h'(s_{c_j^v}(t))\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}}\end{aligned} \tag{18}$$

where $\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}}$ is calculated by

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} y^{\varphi_j}(t) + g(net_{c_j^v}(t)) f'_{in_j}(net_{in_j}(t)) y^m(t-1). \tag{19}$$

Finally, for the weights from input units, memory cell outputs, gates, and standard hidden units to forget gates φ_j,

$$\begin{aligned}\frac{\partial y^K(t)}{\partial w_{\varphi_j m}} = &\sum_{v=1}^{S_j} \Big(w_{K c_j^v} f'_K(net_K(t)) + \sum_h w_{Kh} f'_K(net_K(t)) w_{h c_j^v} f'_h(net_h)(t) \Big) \\ &\cdot y_{out_j}(t) h'(s_{c_j^v}(t)) \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}}\end{aligned} \tag{20}$$

where $\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}}$ is calculated by

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} y^{\varphi_j}(t) + s_{c_j^v}(t) f'_{\varphi_j}(net_{\varphi_j}(t)) y^m(t-1). \tag{21}$$

Because the initial state of the network does not depend on the weights, $\frac{\partial s_{c_j^v}(0)}{\partial w_{im}} = 0$ for all units i that need to store this information, i.e. the CECs, the input gates, and the forget gates.

3.4 Exploration

Non-Markovian RL requires extra attention to the issue of exploration (Chrisman, 1992; McCallum, 1996; Wiering & Schmidhuber, 1997). Undirected exploration attempts to try out actions in the same way in each environmental state. However, in non-Markovian tasks, the agent initially does not know which environmental state it is in. Part of the exploration must be aimed at discovering the environmental state structure. Furthermore, in many cases, the non-Markovian environment will provide unambiguous sensory information indicating the state in some parts, while providing ambiguous sensory information (hidden state) in other parts. In general, we want more exploration in the ambiguous parts.

This paper employs a directed exploration technique based on the combination of these ideas with a kind of temporal differences learning. A separate multilayer feedforward neural network with one output unit y^v is trained, using standard backpropagation, concurrently with the RL-LSTM network. It learns to predict a measure related to the expected *variance* in Advantage values for the current observation, plus its own, discounted prediction at the next timestep, using

$$y_d^v(t) = |T(t)| + \beta y^v(t+1) \tag{22}$$

where $y_d^v(t)$ is the desired value for output $y^v(t)$, and β is a discount parameter in the range $[0,1]$.

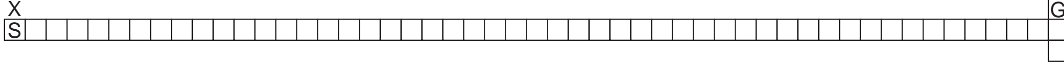

Figure 2: Long-term dependency T-maze with $N = 50$. At the starting position S the agent's observation indicates where the goal position G is in this episode.

Generally speaking, an ambiguous observation for which the various actions differ strongly in value, as well as an observation that precedes such ambiguous observations, will lead to high y^v. y^v is used to linearly scale the temperature of a Boltzmann action selection rule. The Boltzmann action selection rule (Sutton & Barto, 1998; Kaelbling et al., 1996) is modified to have maximum action selection probability of .95, in order to always have at least 5% exploratory actions.

The net effect of the direct exploration mechanism is much exploration when, for the current observation, differences between Advantages are small, or when there is a lot of uncertainty about current Advantages or Advantages in the near future. This exploration technique has obvious similarities with the statistically more rigorous technique of Interval Estimation (Kaelbling, 1990), as well as with certain model-based approaches where exploration is greater when there is more uncertainty in the predictions of a model (Schmidhuber, 1991a; Thrun & Möller, 1992).

4 Test problems

4.1 T-maze

The first test problem is a non-Markovian grid-based T-maze (see figure 2). It was designed to test RL-LSTM's capability to bridge long time lags, without confounding the results by making the control task difficult in other ways. The agent has four possible actions: move North, East, South, or West. The agent must learn to move from the starting position at the beginning of the corridor to the T-junction. There it must move either North or South to a changing goal position, depending on a "road sign" it has observed at the starting position. If the agent takes the correct action at the T-junction, it receives a reward of 4. If it takes the wrong action, it receives a reward of $-.1$. In both cases, the episode ends and a new episode starts, with the new goal position set randomly either North or South. During the episode, the agent receives a reward of $-.1$ when it stands still.

The network has 3 input units, 12 standard hidden units, and 3 memory cells. At the starting position, the observation is either `011` or `110`, at the T-junction the observation is `010`. A noise-free and a very noisy version of the task were investigated. In the noise-free condition, the observation in the corridor is `101`. In the noisy condition, the observation in the corridor is `a0b`, where `a` and `b` are independent, uniformly distributed random values in the range $[0, 1]$, generated online. Different lengths N of the corridor were attempted, varying from 10 to 70. The following parameter values were used in all conditions: $\gamma = .98$, $\lambda = .8$, $\kappa = .1$, $\alpha = .001$.

If the agent takes only optimal actions to the T-junction, it must remember the observation from the starting position for N timesteps to determine the optimal action at the

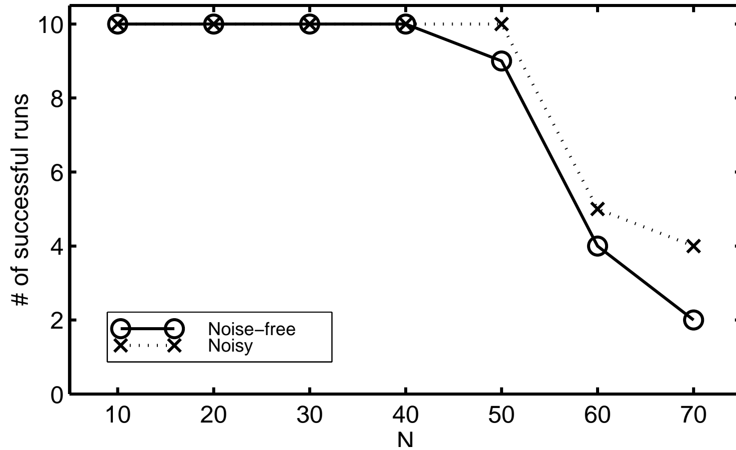

Figure 3: Number of successful runs (out of 10) as a function of N, length of the corridor.

T-junction. Note that the agent is not aided by experiences in which there are shorter time lag dependencies. In fact, the opposite is true. Initially, it takes many more actions until even the T-junction is reached, and the experienced history is very variable from episode to episode. The agent must first learn to reliably move to the T-junction. Because of the long delay and uncertainty of reward, this is already non-trivial, and it is one of the important bottlenecks in achieving good performance. Once this is accomplished, the LSTM network will begin to experience more or less consistent and shortest possible histories of observations and actions, from which it can learn to extract the relevant piece of information. The directed exploration mechanism is crucial in this regard. It learns to set exploration low in the corridor and high at the T-junction. This results in the desired constancy in behavior where it is possible (the corridor), while still maintaining sufficient exploration where it is necessary (the T-junction).

A run was considered a success if the agent learned to take the correct action at the T-junction in over 80% of cases, using its stochastic action selection mechanism. For N up to 40, all runs reached this criterion. In practice, this corresponds to 100% correct action choices at the T-junction using greedy action selection, as well as optimal or near-optimal selection of actions leading to the T-junction. In T-mazes with longer time lags than 40 or 50, the network solves the task only in a slowly decreasing proportion of runs. Figures 3 and 4 summarize the results.

Surprisingly, performance is not worse in the noisy condition than in the noise-free condition; it is even slightly better. An explanation for this somewhat counterintuitive result could be that in the noise-free condition, the system may tend toward reactive policies that directly map observations to actions. In the noisy condition, because of the much greater variety and unpredictability of observations, the system may be less tempted to focus only on the observations as predictors of value, and thus may be "stimulated" to start using its memory cells. In any case, the results show that RL-LSTM is robust in the presence of severe noise.

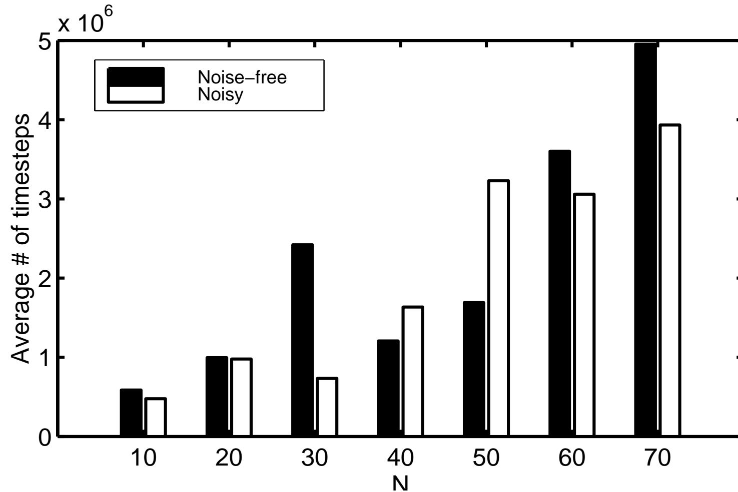

Figure 4: Average number of timesteps until success as a function of N, length of the corridor.

Rylatt and Czarnecki (Rylatt & Czarnecki, 2000) investigate a similar T-maze task, using however the much easier *supervised* learning version of the task. A specially modified recurrent neural network architecture trained using backpropagation through time could only solve the task up to a time lag of at most 13 timesteps. In other supervised timeseries learning studies, backpropagation through time and real-time recurrent learning are similarly known to fail when time lags exceed more than a few timesteps (Bengio et al., 1994; Hochreiter & Schmidhuber, 1997).

On the other hand, supervised LSTM is known to be capable of bridging time lags of over 1000 timesteps in some cases. There are several possible reasons why RL-LSTM does not reach that level of performance, all related to the inherent differences between supervised learning and reinforcement learning. One reason is the fact that, unlike supervised learning, the agent largely determines its history of inputs and outputs itself, as described above. This results in a less constant timeseries than in supervised learning (certainly in the beginning of learning), in which it is harder to find the regularities. Another reason is the fact that in RL, target outputs change over time, which may lead to radically changing backpropagated errors. A third reason is that in those supervised learning tasks where LSTM learned to bridge more than 1000 timesteps, only the outputs at very few timesteps mattered, and error was only injected at those timesteps. In RL, the system must output the correct action at each timestep, and it receives a temporal difference error at each timestep. To illustrate these differences, when the task is modified to a version where the policy leading up to the T-junction is fixed and only the last action needs to be learned using RL, RL-LSTM can easily learn time lag dependencies over 100 timesteps.

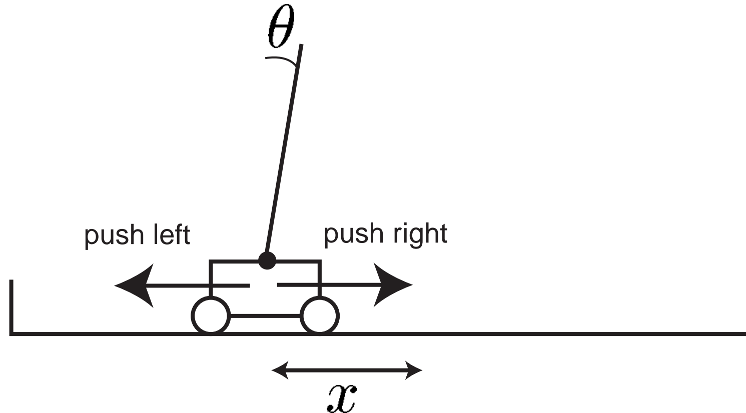

Figure 5: The pole balancing environment.

4.2 Multi-mode pole balancing

The second test problem is a difficult variation of the classical pole balancing task. In the pole balancing or inverted pendulum task (see figure 5), an agent must balance an inherently unstable pole, hinged to the top of a wheeled cart that travels along a track, by applying left and right forces to the cart. The state of the environment is defined by the cart position x, the pole angle θ, the cart velocity $\dot{x}$, and the pole's angular velocity $\dot{\theta}$. Even in the Markovian version, the task requires fairly precise control to solve it.

The version used in this experiment is made more difficult by two sources of hidden state. First, as in (Lin & Mitchell, 1993; Schmidhuber, 1991b), the agent cannot observe the state information corresponding to the cart velocity $\dot{x}$ and pole angular velocity $\dot{\theta}$. It has to learn to approximate this information using its recurrent connections in order to solve the task. Second, the agent must learn to operate in two different modes. In mode A, action 1 is left push and action 2 is right push; in mode B, action 1 is *right* push and action 2 is *left* push. Modes are randomly set at the beginning of each episode. The information which mode the agent is operating in is provided to the agent only for the first second of the episode. After that, the corresponding input unit is set to zero and the agent must *remember* which mode it is in. Obviously, failing to remember the mode leads to very poor performance.

The only reward signal is -1 if the pole falls past $\pm 12°$ or if the cart hits either end of the track. These are also the only points where the episode ends and a new one starts. Note that the agent must learn to remember the mode information for an infinite amount of time if it is to learn to balance the pole indefinitely. This rules out history window approaches altogether. However, in contrast with the T-maze task, the system now has the benefit of starting with relatively short time lags. Furthermore, the task requires the network to organize its internal state space such that it represents both continuous information (cart and pole velocities) and discrete information (mode of operation). Both this requirement and the requirement to remember information indefinitely have been described by Williams (1990) as examples that illustrate the promise of recurrent neural networks for control. It is hard to see how either a system based on traditional control theory (which is mainly concerned

with continuous systems, and which uses fixed size history windows), or a system based on traditional artificial intelligence (mainly concerned with discrete systems) could solve such a task.

The LSTM network has 2 output units, 14 standard hidden units, and 6 memory cells. It has 3 input units: one each for x and θ; and one for the mode of operation, set to zero after one second of simulated time (50 timesteps). $\gamma = .95$, $\lambda = .6$, $\kappa = .2$, α is .0003 for output units and .002 for other units. In this problem, no directed exploration was used, because in contrast to the T-maze, imperfect policies lead to many different experiences with reward signals, and there is hidden state everywhere in the environment.

Out of 10 runs, 2 runs achieved optimal performance (after an average of 6,250,000 timesteps of learning). These two agents were able to balance the pole indefinitely in both modes of operation. In the other 8 runs, the agents still learned to balance the pole in both modes for hundreds or even thousands of timesteps (after an average of 8,095,000 timesteps of learning), thus showing that the mode information was remembered for long time lags. In most cases, such an agent learns optimal performance for one mode, while achieving good but suboptimal performance in the other.

5 Conclusions

The results presented in this paper suggest that reinforcement learning with Long Short-Term Memory (RL-LSTM) is a promising approach to solving non-Markovian RL tasks with long-term dependencies. This was demonstrated in a T-maze task with minimal time lag dependencies of up to 70 timesteps, as well as in a non-Markovian version of pole balancing where optimal performance requires remembering information indefinitely. RL-LSTM's main power is derived from LSTM's property of constant error flow, but for good performance in RL tasks, the combination with Advantage(λ) learning and directed exploration was crucial.

Acknowledgements

I wish to thank Edwin de Jong, Michiel de Jong, and Gwendid van der Voort van der Kleij for valuable comments and discussions.

References

Abul, A., Polat, F., & Alhaji, R. (2000). Multiagent reinforcement learning using function approximation. *IEEE Transactions on Systems, Man, and Cybernetics. Part C: Applications and reviews*, *30*, 485–497.

Bakker, B., & van der Voort van der Kleij, G. (2000). Trading off perception with internal state: Reinforcement learning and analysis of Q-Elman networks in a Markovian task. In *Proceedings of the international joint conference on neural networks, ijcnn'2000* (Vol. 3, pp. 213–218).

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. In *Advances in neural information processing systems 6* (pp. 75–82). San Mateo, CA: Morgan Kaufmann.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the tenth national conference on artificial intelligence.* San Jose, CA: AAAI Press.

Cliff, D., & Ross, S. (1994). Adding temporary memory to ZCS. *Adaptive Behavior*, *3:2*, 101–150.

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Computation*, *12 (1)*, 219–245.

Gers, F., Schmidhuber, J., & Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, *12 (10)*, 2451–2471.

Gomez, F. J., & Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the international joint conference on artificial intelligence.* Denver, CO: Morgan Kaufmann.

Harmon, M. E., & Baird, L. C. (1996). *Multi-player residual advantage learning with general function approximation* (Technical report No. WL-TR-1065). Wright-Patterson Air Force Base Ohio: Wright Laboratory.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9 (8)*, 1735–1780.

Kaelbling, L. P. (1990). *Learning in embedded systems.* PhD thesis, Dept. of Computer Science, Stanford University.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Lanzi, P. L. (2000). Adaptive agents with reinforcement learning and internal memory. In J.-A. Meyer, D. Floreano, H. L. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 6: Proceedings of the sixth international conference on simulation of adaptive behavior* (pp. 333–342). Cambridge, MA: MIT Press.

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

Lin, L.-J., & Mitchell, T. (1993). Reinforcement learning with hidden states. In J.-A. Meyer, H. L. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 2: Proceedings of the second international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Littman, M. L. (1993). An optimization-based categorization of reinforcement learning environments. In J.-A. Meyer, H. L. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 2: Proceedings of the second international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Littman, M. L. (1994). Memoryless policies: theoretical limitations and practical results. In D. Cliff, P. Husbands, J.-A. Meyer, & S. W. Wilson (Eds.), *From animals to animats 3: Proceedings of the third international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research*, *28*, 47–66.

McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the tenth international machine learning conference.*

McCallum, R. A. (1996). Learning to use selective attention and short-term memory in sequential tasks. In *From animals to animats 4: Proceedings of the fourth international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Meuleau, N., Peshkin, L., Kim, K. E., & Kaelbling, L. P. (1999). Learning finite-state controllers for partially observable environments. In *Proceedings of the fifteenth conference on uncertainty in artificial intelligence.*

Peshkin, L., Meuleau, N., & Kaelbling, L. P. (1999). Learning policies with external memory. In *Proceedings of the sixteenth international conference on machine learning.*

Rylatt, R. M., & Czarnecki, C. A. (2000). Embedding connectionist autonomous agents in time: The 'road sign problem'. *Neural Processing Letters*, *12*, 145–158.

Schmidhuber, J. (1991a). Curious model-building control systems. In *Proceedings of the international joint conference on neural networks, IJCNN'91* (Vol. 2, pp. 1458–1463). Singapore.

Schmidhuber, J. (1991b). Reinforcement learning in Markovian and non-Markovian environments. In D. S. Touretzky (Ed.), *Advances in neural information processing systems 3* (pp. 500–506). San Mateo, CA: Morgan Kauffman.

Suematsu, N., & Hayashi, A. (1999). A reinforcement learning algorithm in partially observable environments using short-term memory. In *Advances in neural information processing systems* (Vol. 11, pp. 1059–1065).

Sutton, R. S. (1989). *Implementation details of the TD(λ) procedure for the case of vector predictions and backpropagation* (Technical report No. TN87-509.1). GTE Laboratories.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* Cambridge, MA: MIT Press.

Thrun, S. B., & Möller, K. (1992). Active exploration in dynamic environments. In *Advances in neural information processing systems 4.* San Mateo, CA: Morgan Kaufmann.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* PhD thesis, Cambridge University.

Wiering, M., & Schmidhuber, J. (1997). HQ-learning. *Adaptive Behavior*, *6:2*, 219–246.

Williams, R. J. (1990). Adaptive state representation and estimation using recurrent connectionist networks. In W. T. Miller, R. S. Sutton, & P. J. Werbos (Eds.), *Neural networks for control.* Cambridge, MA: MIT Press.