**Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)**

# Reinforcement learning by backpropagation through an LSTM model/critic

Bram Bakker
Intelligent Systems Laboratory Amsterdam
Informatics Institute, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
Email: bram@science.uva.nl

*Abstract*— **This paper describes backpropagation through an LSTM recurrent neural network model/critic, for reinforcement learning tasks in partially observable domains. This combines the advantage of LSTM's strength at learning long-term temporal dependencies to infer states in partially observable tasks, with the advantage of being able to learn high-dimensional and/or continuous actions with backpropagation's focused credit assignment mechanism.**

## I. Introduction

There are multiple reasons why neural networks (NNs) are used for approximate dynamic programming or reinforcement learning (RL) problems. Probably the most common reason is that the particular RL problem under consideration has a *large and/or continuous state space*, and that generalization over the state space is required after sampling only parts of the state space. Likewise, the RL problem may have a *large and/or continuous action space*, such that the actions can no longer be enumerated as in standard RL (such as standard table-based Q-learning), but instead a high-dimensional, and possibly continuous-valued action vector is considered. A third reason may be *partial observability*, or the problem of hidden state, which means that the observations (sensor readings) provide only noisy, limited or ambiguous information about the underlying (Markov) state.

This paper focuses on the difficult case where all three issues play a role: a large and/or continuous state space, a large and/or continuous action space, and partial observability. We describe how this case may be handled with *backpropagation through a model* (BTM), where the model is a recurrent neural network (RNN). In particular, we use the Long Short-Term Memory (LSTM) architecture, because it has in previous work been successful even in hard hidden state problems.

The next section describes different approaches to combining RL with NNs. Section III describes LSTM and backpropagation through LSTM. Section IV describes some initial experimental results. Section V, finally, presents conclusions.

## II. Combining RL with NNs

Different approaches to combining RL with NNs can be taken (see [17], [3] for overviews), and the approach depends to a considerable extent on the reason to use neural networks. In the following, first the basics of RL as well as some notation are discussed. Next, approaches for completely observable RL problems are discussed, in which feedforward neural networks can be used. Finally, approaches for partially observable RL problems are discussed, in which recurrent neural networks can be used, and the approach described in this paper, backpropagation through an LSTM model/critic, is introduced.

### A. Reinforcement learning

In RL, the learning system (agent, robot, controller) interacts with an environment and attempts to optimize an objective function based on scalar reward (or cost) signals (see [15] for an overview of RL, using the same notation as the one used here). Its *actions* $a$ (controls) modify its *state* in the environment $s$, and also lead to immediate *rewards* $r$. *Observations* $o$ provide the learning system with information about $s$.

Generally, the objective of RL (in the discrete time case) is to determine a policy $\pi$ which at time $t$ selects an action $a_t$ given the state $s_t$ and which maximizes the expected discounted future cumulative reward: $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_i \gamma^i r_{t+i}$, where $\gamma \in [0, 1]$ is a factor which discounts future rewards. Many RL algorithms estimate a utility or *value function*, which approximates this measure of expected discounted future cumulative reward, and which may be defined over the state space ($V(s)$) or the state-action space ($Q(s, a)$).

If a discrete set of states and actions is considered, and the observations $o$ are equivalent to states $s$, the problem can be formulated as a Markov Decision Process (MDP). If there is hidden state or partial observability, i.e. the observations are not equivalent to states but contain partial or incomplete information about the state, the problem can be formulated as a Partially Observable Markov Decision Process (MDP). In this paper we consider the more general case of possibly continuous states, observations, and actions, as well as the issue of partial observability (but we do not consider continuous time).

### B. Completely observable RL problems

**Direct value function approximation.** If the state space is large and/or continuous but completely observable and the action space is relatively small, discrete, and enumerable, an NN such as a multilayer feedforward neural network may be used as a function approximator, to approximate the value function of a temporal difference-based reinforcement learning

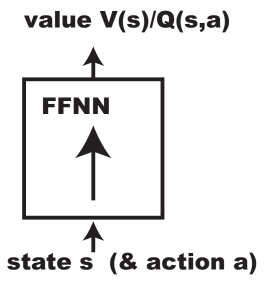

Fig. 1. Direct value function approximation approach for completely observable problems: a feedforward neural network is used as function approximator for an RL value function.

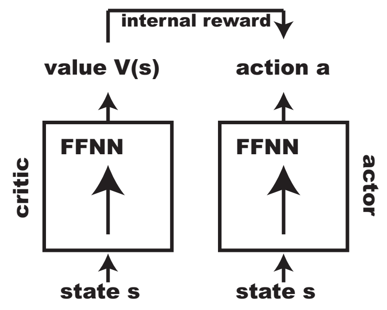

Fig. 2. Actor-critic approach for completely observable problems: one feedforward neural network, the critic, is used as function approximator for an RL value function; another, the actor, to learn the policy.

algorithm (see figure 1). The input vector contains state information, the output represents values of the RL value function. This is the approach taken, for example, in Tesauro's well-known world class backgammon player [16]. In this approach a single output unit may be used to compute values, and the network may be recomputed for different possible actions (as in Tesauro's work). If a value function over the state-action space (like Q-learning) rather than the state space is to be computed, the inputs to the network may include a code for different actions, or different output units may represent values of different actions, or multiple networks may represent different actions. A variation of Q-learning may be used for training the network. In that case, the desired value $d_k$ at time $t$ for the corresponding output unit $k$ is:

$$d_k(t) = r(t) + \gamma \max_a Q(s(t+1), a) \quad (1)$$

where $r(t)$ is the immediate reward achieved after executing action $a(t)$ in state $s(t)$ and reaching state $s(t+1)$, $Q(s,a)$ is the state-action value function representing cumulative discounted reward (estimated by the network), and $\gamma$ is a discount factor. After learning, in each state the best action can then be selected by simply taking that action which leads to the highest value computed by the network(s).

**Actor-critic system.** If the state space is large and completely observable, but the action space is large and/or continuous, then an actor-critic architecture (see figure 2) may be used in which, as above, a network approximates a value function. This network is called the critic. Now, however, there is also a second network, called the actor. Like the critic, the actor receives as input state information. Its output is a vector coding for the action (or action probabilities). The actor is trained using "internal rewards" from the critic, e.g. positive (negative) internal rewards when its actions lead to positive (negative) temporal difference errors for the critic.

**Backpropagation through a model/critic.** One disadvantage of standard actor-critic systems is that search for the best action vector is undirected and can therefore be slow and fail to converge to good solutions, as it is based on the actor performing a kind of stochastic hillclimbing in the action space. The backpropagation through a model/critic (BTM) approach can remedy this, as it exploits backpropagation's

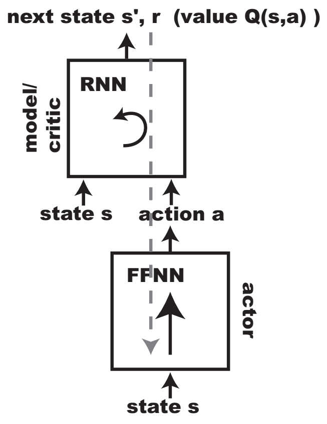

Fig. 3. Backpropagation through a model/critic approach for completely observable problems: one neural network is used as a predictive model of the environment. In addition, this network may estimate a value function. Another neural network, the actor, learns the policy based on errors backpropagated through the (frozen) model/critic (dotted lines).

directed credit assignment mechanism to learn the action (see figure 3). The idea is that rather than using an NN to directly represent an RL value function, an NN is used to learn a model of the environment that predicts next observations and rewards based on current observations and actions. This network is typically a recurrent neural network. Stochastic RL problems can be dealt with as well, if the outputs of the model are interpreted as probabilities.

Additionally or alternatively, the model network may learn to predict a value function, based on a standard temporal difference-based reinforcement learning algorithm. This network is called the model/critic [17].

A separate neural network, the actor network, learns mappings from its input vector to an action vector, which is connected to the input side of the model. In the actor-learning phase of BTM, the weights of the model are fixed, and errors are backpropagated through the model that reflect not prediction errors (the model is assumed to be accurate or already learned), but differences between *desired* outcomes in the environment and *actual* outcomes. These errors are

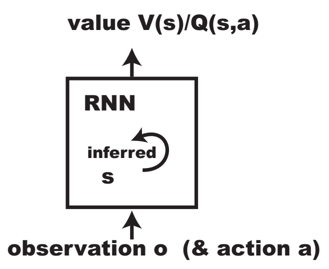

Fig. 4. Direct value function approximation approach for partially observable problems: a recurrent neural network is used as function approximator for an RL value function.

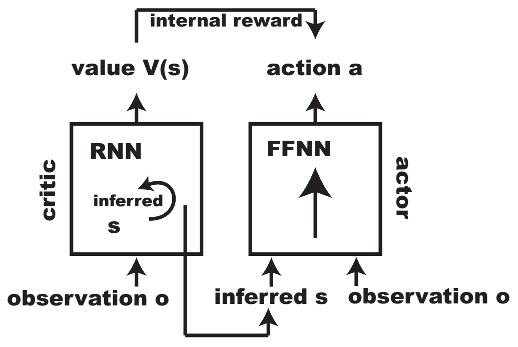

Fig. 5. Actor-critic approach for partially observable problems.

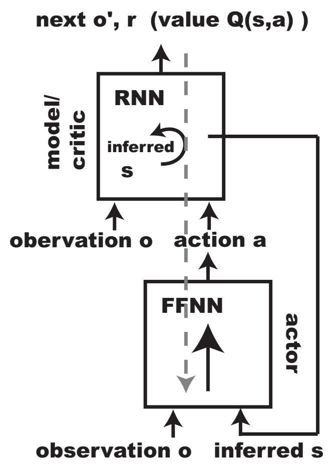

Fig. 6. Backpropagation through a model/critic approach for partially observable problems: one recurrent neural network is used as a predictive model of the environment. In this paper, this is an LSTM network. In addition, this network may estimate a value function. Another neural network, the actor, learns the policy based on errors backpropagated through the (frozen) model/critic (dotted lines). The actor receives the state inferred by the model/critic as (additional) input.

backpropagated to the input side of the model, where they represent how the inputs, in particular the inputs representing actions, should change so as to improve the outcomes in the environment. The actor then learns the correct actions on the basis of these errors in a supervised learning way, again using backpropagation.

In the supervised learning version of backpropagation through a model [17], [7], [8], [11], desired outputs of the model (desired environmental states) are used to determine the errors that are to be backpropagated to the actor. In the reinforcement learning case [14], an output of the model is reward. The "desired output" then corresponds to "high reward" on this output unit, which can be backpropagated through time to past input activations representing actions. If the model is (also) a critic and thus estimates a value function on one of the output units, similarly desired outputs correspond to high values on this output. Because a value function explicitly represents future cumulative discounted rewards available from individual state onwards, this may make temporal credit assignment easier and help in obtaining high cumulative discounted rewards. See [12] for an extensive discussion of the relationship between BPTT and temporal credit assignment based on value functions.

An elegant property of the BTM approach is that both the model and the actor can be neural networks, and a single procedure, backpropagation, can be used to adjust the weights of the model, to compute the errors for the actor, and to adjust the weights of the actor. A disadvantage, on the other hand, is that this approach can be sensitive to small errors in the model, which may lead to large errors in the signals propagated back to the actions [7], [17], [3].

### C. Partially observable RL problems

**Direct value function approximation.** If the action space is relatively small but the state space is partially observable, a direct value function approximation approach may be used (see figure 4) based on *recurrent* neural networks (RNNs) [9]. However, in this case the state is not given. The system makes observations containing *partial* information about the state. The interesting element here is that the RNNs must now not only learn to approximate the value function, but at the same time learn to infer the environment's state based on the recurrent activations representing the history of observations and actions. Desired, target outputs are provided by a temporal difference RL algorithm (such as Q-learning). Essentially, temporal patterns in the observed sequence of observations and actions must be discovered which allow the network to correctly estimate the value functions. A problem here is that in general learning temporal patterns in timeseries data is difficult, especially if there are long-term dependencies between inputs and outputs that must be discovered [4]. In previous work, we showed that a Long Short-Term Memory (LSTM) RNN [6] is capable of this in an RL context even if it must learn to detect complex and long-term temporal dependencies between past and current observations and actions to infer the correct state [1].

**Actor-critic system.** Actor-critic systems can be used in partially observable domains (see figure 5), but as in the the completely observable case it suffers from the problem that search for the best action vector is undirected and can be slow, being based on a kind of stochastic hillclimbing in the action space.

**Backpropagation through a model/critic.** If the state space is large and/or continuous and only partially observable, *and* the action space is large and/or continuous, the BTM approach based on recurrent neural networks is one of the few options to handle this very difficult case [13], [17].

As in the completely observable case, one neural network serves as a model/critic, learning to predict next observations and rewards and/or values (see figure 6). In the partially observable case however, this modeling task is significantly more difficult as it only sees observations with partial information about the state. As in the direct value function approximation approach to partially observable problems, a recurrent neural network must be used which learns to infer the state based on the experienced sequence of observations, actions, and rewards. For the actor, as well, the task is more difficult because it has no access to the complete state. One can use the state as inferred by the model/critic as input to the actor. The actor can then learn the mapping from the inferred state to actions, again based on errors backpropagated through the model/critic.

In this paper we describe how the BTM approach can be realized with the LSTM architecture in partially observable domains. The idea is that this combines the advantage of LSTM's strength at learning long-term temporal dependencies to infer states in partially observable tasks, with the advantage of being able to learn high-dimensional and/or continuous actions with backpropagation's focused credit assignment mechanism. Both of these advantages reflect important issues in RL, for which there are very few alternatives to (recurrent) neural networks. To the best of our knowledge, this is the first time the LSTM architecture is combined with the BTM approach, either in a supervised learning context or reinforcement learning context.

## III. Backpropagation through an LSTM model/critic

### A. The LSTM architecture

LSTM is a recurrent neural network architecture, originally designed for supervised timeseries learning [6]. Figure 7 shows a typical network. It is based on an analysis of the problems that conventional recurrent neural networks and their corresponding learning algorithms, e.g. simple recurrent networks with standard one step backpropagation [5], or recurrent networks with backpropagation through time (BPTT) [17] or real-time recurrent learning (RTRL), 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 [4], [6]).

LSTM's solution to the problem of vanishing errors with respect to past activations is to enforce *constant* error flow in a number of specialized units, called Constant Error Carrousels (CECs). Access to and from them is regulated using other learning, specialized multiplicative units, called input gates, output gates, and forget gates. The combination of a CEC with its associated input, output, and forget gate is called a *memory*

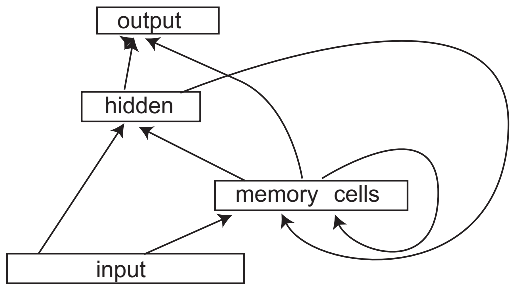

Fig. 7. Typical LSTM network. Each layer consists of multiple units. Arrows indicate unidirectional, fully connected weights.

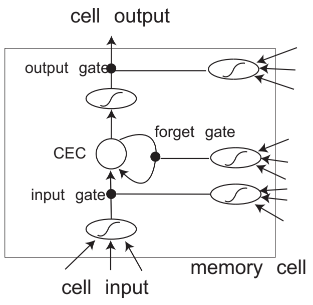

Fig. 8. One memory cell. Access to and from the Constant Error Carrousel (CEC) is regulated using a specialized, multiplicative input gate, output gate, and forget gate.

*cell*, and typical networks (see figure 7) have a number of them. See figure 8 for a schematic of a memory cell.

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

$$net_i(t) = \sum_m w_{im} y_m(t-1) \tag{2}$$

where $w_{im}$ is the weight of the connection from unit $m$ to unit $i$.[1] 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 $y_\varphi$ is computed as

$$y_i(t) = f_i(net_i(t)) \tag{3}$$

where $f_i$ is the standard logistic sigmoid function, squashing the net input to the range $[0,1]$. The CEC activation $z_{c_j^v}$, or the state of memory cell $v$ in memory block $j$, is computed as follows:

$$z_{c_j^v}(t) = y_{\varphi_j}(t) z_{c_j^v}(t-1) + y_{in_j}(t) g(net_{c_j^v m}(t)) \tag{4}$$

where $g$ is a logistic sigmoid function scaled to the range $[-2,2]$, and $z_{c_j^v}(0) = 0$. 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

[1]For the purpose of notation, inputs to the network are viewed as activations of units from one timestep ago, so they can be described in the same way as recurrent activations.

gates, and the memory cell outputs. The forget gate resets the activation of the CECs (in a gradual way) when the information stored in the memory cell is no longer useful. The memory cell's output $y_{c_j^v}$ is calculated by

$$y_{c_j^v}(t) = y_{out_j}(t) h(z_{c_j^v}(t)) \tag{5}$$

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 to other units, among which are the output units. An output unit's activation $y_k$, finally, is computed using

$$y_k(t) = f_k(net_k(t)) = net_k(t). \tag{6}$$

In the architecture used here, output units receive input from all normal hidden units and from all memory cell outputs.

## *B. Learning an LSTM model/critic for BTM*

When using LSTM as the model in a BTM approach (see figure 6), one must first have a phase in which the LSTM model is learned. This can be done by following an exploration policy and using LSTM's learning algorithm [6], [1]. Based on the observations and actions selected by the exploration policy, the next observation and reward are observed, which the LSTM model learns to predict. The exploration policy can be completely random, but in many cases it would be better to follow a policy which focuses on somewhat reasonable trajectories.

An interesting variation (described above, and explored below), is to let the LSTM model learn, in addition to or instead of the next observation and reward, a value function. In that case, we call it an LSTM model/critic. The advantage of that approach is that the value function takes care of part of the temporal credit assignment problem. At the same time, we still have the advantage of being able to backpropagate error vectors toward action vectors. One must still use an RNN (such as LSTM) for the model/critic if we do backpropagation through a critic, because the state is partially observable. Note that even if the LSTM network learns a value function, it is often still useful to also learn to predict the next observation and reward. Using the next observation and reward as additional, supervised training information will in many cases help to infer a good approximation to the state (see [9]).

Learning the weights of an LSTM network is done using an efficient version of Real-Time Recurrent Learning (RTRL), i.e. a variation of the backpropagation algorithm for RNNs [6]. Variations of this can be used in an RL context to learn the value function [1] based on a variation of Q-learning (see eq. 1).

## *C. Real-time recurrent backpropagation through LSTM*

After training the LSTM model/critic, the actor can be trained. It is natural to use the state as inferred by the model as input to the actor. The actor can then learn the mapping from the inferred state to actions.

For BTM, we must compute the partial derivatives of the error on the output side of the model (in this case an LSTM network) with respect to the input units of model, in particular the input units coding for the action.

Thus, we wish to compute $\frac{\partial E(t)}{\partial y_n}$, where $E(t)$ is the error measure at time $t$, and $y_n$ is the activation of input unit $n$. Let us assume $E$ is the standard sum of squared errors, then

$$\frac{\partial E(t)}{\partial y_k(t)} = -2(d_k(t) - y_k(t)) \tag{7}$$

where $d_k(t)$ is the "desired" or target value for output unit activation $y_k(t)$. This leaves us with the task of computing $\frac{\partial y_k(t)}{\partial y_n}$, the partial derivatives of network outputs with respect to input unit activations.

In the variation of RTRL used for learning the LSTM weights, most partial derivatives of errors with respect to weights are truncated, in the sense that recurrent activations from more than one timestep ago are not taken into account (similar to how standard backpropagation can be applied to simple recurrent networks [5]). This happens everywhere except for the connections feeding into the memory cells [6], where the important memory for past events is located. We use the same principle for our Real-Time Recurrent Backpropagation Through a Model (RTR-BTM) algorithm, but in our case gradients of errors with respect to *presynaptic activations* are truncated, such that recurrent activations from more than one timestep ago are not taken into account except for the ones feeding into the memory cells. This truncation allows us to limit the computation and storage requirements, while not losing important memory for past events because the important recurrent connections are within the memory cells where the partial derivatives are not truncated.

Let us introduce the notation $\frac{\partial y_k(t)}{\partial y_{im}}$ to indicate the partial derivative of an output unit $k$ with respect to unit $m$ as a result of intermediate unit $i$ via weight $w_{im}$. This allows us to individually compute the partial derivatives backpropagated via those weights $w_{im}$ and later compute $\frac{\partial y_k(t)}{\partial y_m}$ by summing over all $\frac{\partial y_k(t)}{\partial y_{im}}$. More formally:

$$\begin{aligned}\frac{\partial E(t)}{\partial y_m} &= \frac{\partial E(t)}{\partial y_k(t)} \frac{\partial y_k(t)}{\partial y_m} = \frac{\partial E(t)}{\partial y_k(t)} \sum_i \frac{\partial y_k(t)}{\partial y_i} \frac{\partial y_i}{\partial y_m} \\ &= \frac{\partial E(t)}{\partial y_k(t)} \sum_i \frac{\partial y_k(t)}{\partial y_{im}}.\end{aligned} \tag{8}$$

Thus, we are left with the task of computing $\frac{\partial y_k(t)}{\partial y_{im}}$ for all units $i$ and $m$. They can be derived from the activation update equations by repeated application of the chain rule.

This yields the following equations. Without loss of generality, the specific equations described below are for the case of the network architecture depicted in figure 7. For the weights from memory cell outputs and standard hidden units to output units ($i = k$),

$$\frac{\partial y_k(t)}{\partial y_{km}} = f'_k(net_k(t)) w_{km}. \tag{9}$$

For the memory cell outputs and input units feeding into standard hidden units $h$,

$$\frac{\partial y_k(t)}{\partial y_{hm}} = w_{kh} f'_k(net_k(t)) f'_h(net_h(t)) w_{hm}. \tag{10}$$

For the input units, memory cell outputs, gates, and standard hidden units feeding into output gate units $out_j$,

$$\begin{aligned}\frac{\partial y_k(t)}{\partial y_{out_j m}} =& \sum_{v=1}^{Z_j} h(z_{c_j^v}(t)) \Big( w_{kc_j^v} f'_k(net_k(t)) \\ &+ \sum_h w_{kh} f'_k(net_k(t)) w_{hc_j^v} f'_h(net_h)(t) \Big) \\ &\cdot f'_{out_j}(net_{out_j}(t)) w_{out_j m}\end{aligned} \tag{11}$$

where $Z_j$ is the number of CECs in memory block $j$. For the input units, memory cell outputs, gates, and standard hidden units feeding into CEC units $c_j^v$,

$$\begin{aligned}\frac{\partial y_k(t)}{\partial y_{c_j^v m}} =& \Big( w_{kc_j^v} f'_k(net_k(t)) \\ &+ \sum_h w_{kh} f'_k(net_k(t)) w_{hc_j^v} f'_h(net_h)(t) \Big) \\ &\cdot y_{out_j}(t) h'(z_{c_j^v}(t)) \frac{\partial z_{c_j^v}(t)}{\partial y_{c_j^v m}}\end{aligned} \tag{12}$$

where $\frac{\partial z_{c_j^v}(t)}{\partial y_{c_j^v m}}$ is updated as follows:

$$\frac{\partial z_{c_j^v}(t)}{\partial y_{c_j^v m}} = \frac{\partial z_{c_j^v}(t-1)}{\partial y_{c_j^v m}} y_{\varphi_j}(t) + g'(net_{c_j^v}(t)) y_{in_j}(t) w_{c_j^v m}. \tag{13}$$

For the input units, memory cell outputs, gates, and standard hidden units feeding into input gates $in_j$,

$$\begin{aligned}\frac{\partial y_k(t)}{\partial y_{in_j m}} =& \sum_{v=1}^{Z_j} \Big( w_{kc_j^v} f'_k(net_k(t)) \\ &+ \sum_h w_{kh} f'_k(net_k(t)) w_{hc_j^v} f'_h(net_h)(t) \Big) \\ &\cdot y_{out_j}(t) h'(z_{c_j^v}(t)) \frac{\partial z_{c_j^v}(t)}{\partial y_{in_j m}}\end{aligned} \tag{14}$$

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

$$\begin{aligned}\frac{\partial z_{c_j^v}(t)}{\partial y_{in_j m}} =& \frac{\partial z_{c_j^v}(t-1)}{\partial y_{in_j m}} y_{\varphi_j}(t) \\ &+ g(net_{c_j^v}(t)) f'_{in_j}(net_{in_j}(t)) w_{in_j m}.\end{aligned} \tag{15}$$

Finally, for the input units, memory cell outputs, gates, and standard hidden units feeding into forget gates $\varphi_j$,

$$\begin{aligned}\frac{\partial y_k(t)}{\partial y_{\varphi_j m}} =& \sum_{v=1}^{Z_j} \Big( w_{kc_j^v} f'_k(net_k(t)) \\ &+ \sum_h w_{kh} f'_k(net_k(t)) w_{hc_j^v} f'_h(net_h)(t) \Big) \\ &\cdot y_{out_j}(t) h'(z_{c_j^v}(t)) \frac{\partial z_{c_j^v}(t)}{\partial y_{\varphi_j m}}\end{aligned} \tag{16}$$

where $\frac{\partial z_{c_j^v}(t)}{\partial y_{\varphi_j m}}$ is calculated by

$$\frac{\partial z_{c_j^v}(t)}{\partial y_{\varphi_j m}} = \frac{\partial z_{c_j^v}(t-1)}{\partial y_{\varphi_j m}} y_{\varphi_j}(t) + z_{c_j^v}(t-1) f'_{\varphi_j}(net_{\varphi_j}(t)) w_{\varphi_j m}. \tag{17}$$

The only information that must be stored in this RTR-BTM algorithm for LSTM are the partial derivatives $\frac{\partial z_{c_j^v}(0)}{\partial y_{im}}$ described above. Because the initial state of the network does not depend on the weights, $\frac{\partial z_{c_j^v}(0)}{\partial y_{im}} = 0$ for all units $i$ that need to store this information, i.e. the CECs, the input gates, and the forget gates.

Finally, we can determine $\frac{\partial E(t)}{\partial y_n}$, the partial derivatives of the output errors with respect to input units $n$, by summing over all partials

$$\begin{aligned}\frac{\partial E(t)}{\partial y_n} =& \sum_k \frac{\partial E(t)}{\partial y_k(t)} \Big( \sum_h \frac{\partial y_k(t)}{\partial y_{hn}} + \sum_{out_j} \frac{\partial y_k(t)}{\partial y_{out_j n}} \\ &+ \sum_{in_j} \frac{\partial y_k(t)}{\partial y_{in_j n}} + \sum_{c_j^v} \frac{\partial y_k(t)}{\partial y_{c_j^v n}} + \sum_{\varphi_j} \frac{\partial y_k(t)}{\partial y_{\varphi_j n}} \Big).\end{aligned} \tag{18}$$

These are the backpropagated errors to the input side of the LSTM model/critic network. For those input units that represent the action vector, these backpropagated errors yield the training errors for the separate actor neural network. In our experiments, the actor neural network is a multilayer feedforward neural network, and it is trained using standard supervised backpropagation learning based on these errors.

## IV. Example experiment

### A. Learning problem and architecture

We have done several experiments, focusing initially on conceptually simple, artificial RL problems in which the resulting models and policies are easy to analyze and in which the problems of multi-dimensional, continuous action vectors as well as partial observability with long-term temporal dependencies play a role and can be varied.

In our example experiment, the LSTM model also learns a value function while learning the model of the environment, and thus is an LSTM model/critic. A separate multilayer feedforward neural network, the actor network, learns mappings from its input vectors to action vectors based on errors that are backpropagated through the LSTM model/critic. The actor network's inputs consist of the observations, which represent a partial view on the partially observable state, and the LSTM model/critic network's internal state, i.e. its memory cell activations, which are supposed to capture the environment state inferred by the LSTM model/critic.

Our example problem is a variation of an artificial "navigation problem" which we as well as others studied before [10], [1], [2]. In the variation of the problem considered here, the learning system must take sequences of actions to a goal state. The task is deterministic but partially observable. The observation is a 3-dimensional continuous vector, the action is a 5-dimensional continuous vector. The learning system has at its disposal discrete movement actions "North", "East", "South", or "West", each of which is represented by one element in its action vector, and which can bring the learning system closer to the goal state, at which point it can receive a large reward. But this reward also depends on a continuous element in its action vector. The maximum final reward is 5. There are no other rewards in the task, but the rewards are discounted, using discount factor $\gamma = .99$, meaning that there is an incentive to reach the goal as quickly as possible.

The difficulty of the task lies in particular in the fact that its rewards depend on an observation vector which it can only see at the start state and the important elements of which it must remember until it reaches the goal state—this makes the problem partially observable. That important observation at the start state consists of discrete and continuous elements. The discrete elements determine the final action to reach the goal state. The continuous elements determine the best value of the continuous element in the action vector at the goal state.

The problem can be varied such that the minimum number of actions until the goal state is reached is varied, and with that the minimum number of timesteps that the system must remember this observation (making it more or less difficult to learn this temporal pattern). In the example experiment described here, this number is 20, meaning that the LSTM model/critic must learn to temporal dependencies bridging at least 20 timesteps. If non-optimal actions are taken between start state and goal state, the system must remember this observation for a longer time. Note that detecting and learning temporal dependencies bridging 20 timesteps or more and remembering information for that long is difficult for most RNNs.

In addition, the observation vectors between the start state and the goal state are very noisy; uniformly distributed noise is added to these observations. This means that both the LSTM model network and the actor must be robust under noisy perturbations of the inputs.

In summary, discrete and continuous information must be remembered for extensive periods of time, requiring a robust mechanism for learning long-term dependencies to infer the state; and a continuous-valued multi-dimensional action vector must be learned, requiring a mechanism like BTM.

### B. Training and results

**LSTM model/critic learning.** The first learning phase is the LSTM model/critic learning phase. In this phase, an exploration policy takes actions, and the LSTM model/critic is trained, using LSTM's normal RTRL algorithm (see section III-B), to predict the next observation, the next reward, and the value of the value function. The value function is learned based on temporal difference Q-learning (eq. 1).

Note that Q-learning is an off-policy RL method (see [15]), meaning that a value function is learned which is independent of the specific exploration policy (given that there is sufficient exploration). Off-policy value function learning is necessary, as the final policy learned by the actor differs from the exploration policy used here, but must make use of the same value function. The exploration policy in this case was a heuristic policy based on estimating the state-action value with the LSTM model/critic a number of times using different random action vectors, and selecting one of those action-vectors using $\epsilon$-greedy exploration.

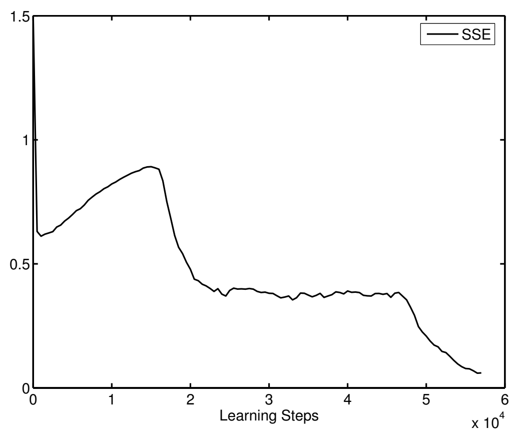

Fig. 9. Sum of Squared Errors (SSE) achieved by the LSTM model/critic as a function of learning steps (the first learning phase).

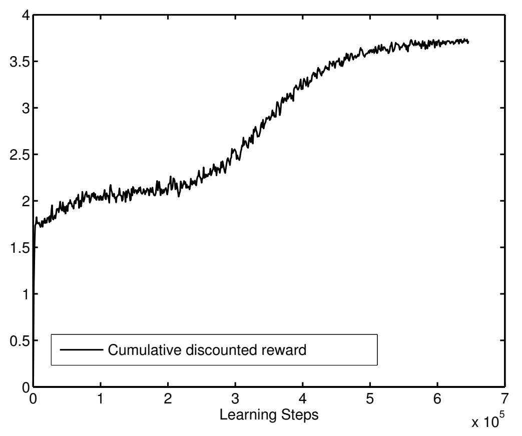

Fig. 10. Cumulative discounted reward achieved by the actor as a function of learning steps (the second learning phase).

As an illustration, figure 9 shows a running average of the Sum of Squared Errors (SSE) of a typical run of the LSTM model/critic during its learning process. It is interesting to note that the SSE initially increases for a period of time. We hypothesize that this is caused by the developing value function. Eventually, however, the SSE decreases to a low level, and an accurate model is learned. Note that this implies that long-term temporal regularities have been learned to correctly infer the state, which allows the model to accurately predict next observations and rewards.

**Actor learning.** After the LSTM model/critic has been trained, the actor learning phase starts. Now, the standard exploration policy is turned off, and exploration is continued by the actor. It is trained by using standard backpropagation. Its desired outputs (target values) are obtained by applying BTM through the trained, fixed LSTM model/critic, with desired outputs for the value function output unit corresponding to high values.

Figure 10 shows a running average of the cumulative discounted reward (which the system is attempting to optimize) obtained by the actor as a function of the number of learning steps in a typical run (the second learning phase). It is apparent that in the course of the learning process, the performance gradually increases. Final performance is near-optimal, as optimal behavior corresponds to a cumulative discounted reward of 4.09.

## V. Conclusions and discussion

In this paper we have described backpropagation through LSTM models, and applied it to reinforcement learning under partial observability of the state. An efficient algorithm, Real-Time Recurrent Backpropagation Through a Model (RTR-BTM), backpropagates errors from the output side of the LSTM model/critic to the action vector of a separate actor, allowing the latter to learn high-dimensional and/or continuous actions using the focused credit assignment of backpropagation. Experimental results in test problems demonstrate the feasibility of the approach.

An interesting possibility for future research would be to investigate procedures to go back and forth between the model/critic learning phase and the actor learning phase. This would allow the model/critic to improve its predictions of next observations and rewards (and the value function) based on the latest policy learned by the actor. The reason is that the latest policy learned by the action will, if it is already reasonable, help in focusing exploration on the most interesting parts of the state-action space and help in getting better estimates of how much reward can be obtained. The model/critic improved in this way may then be used to improve the policy learned by the actor, and so on.

In general, the most interesting part of the approach discussed in this paper may be that it allows for a relatively focused search for good high-dimensional and/or continuous action vectors, without having to resort to supervised learning. There may be other ways of doing such a focused search for good high-dimensional action vectors, but it seems clear that that is what is needed for many interesting real-world RL problems.

## References

[1] B. Bakker. Reinforcement learning with Long Short-Term Memory. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*. MIT Press, Cambridge, MA, 2002.

[2] B. Bakker, V. Zhumatiy, G. Gruener, and J. Schmidhuber. A robot that reinforcement-learns to identify and memorize important previous observations. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 430–435, 2003.

[3] Andrew G. Barto. Connectionist learning for control. In W. Thomas Miller, Richard S. Sutton, and Paul J. Werbos, editors, *Neural networks for control*, chapter 1, pages 5–58. M.I.T. Press, Cambridge, Mass, 1990.

[4] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. In *Advances in Neural Information Processing Systems 6*, pages 75–82, San Mateo, CA, 1994. Morgan Kaufmann.

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

[6] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8):1735–1780, 1997.

[7] M. I. Jordan and D. E. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:370–354, 1992.

[8] M. Kawato. Computational schemes and neural network models for formation and control of multi-joint arm trajectory. In W. Thomas Miller, Richard S. Sutton, and Paul J. Werbos, editors, *Neural networks for control*. M.I.T. Press, Cambridge, Mass, 1990.

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

[10] 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 *Proceedings of the International joint Conference on Artificial Intelligence, IJCAI'2001*, 2001.

[11] D. Nguyen and B. Widrow. The truck backer-upper: An example of self-learning in neural networks. In W. Thomas Miller, III, Richard S. Sutton, and Paul J. Werbos, editors, *Neural Networks for Control*, chapter 12, pages 287–299. MIT Press, 1990.

[12] D. Prokhorov. Backpropagation through time and derivative adaptive critics: A common framework for comparison. In J. Si et al., editor, *Learning and Approximate Dynamic Programming*. Wiley, 2004.

[13] J. Schmidhuber. Networks adjusting networks. In *Proc. of Distributed Adaptive Neural Information Processing*, St. Augustin, 1990.

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

[15] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA, 1998.

[16] G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[17] Paul J. Werbos. A menu of designs for reinforcement learning over time. In W. Thomas Miller, Richard S. Sutton, and Paul J. Werbos, editors, *Neural networks for control*, chapter 3, pages 67–95. M.I.T. Press, Cambridge, Mass, 1990.