A Scalable Model-Free Recurrent Neural Network Framework for Solving POMDPs

Zhenzhen Liu, *Student Member, IEEE*
Department of Electrical & Computer Engineering
University of Tennessee
Knoxville, TN 37996-2100

Itamar Elhanany, *Senior Member, IEEE*
Department of Electrical & Computer Engineering
University of Tennessee
Knoxville, TN 37996-2100

Abstract— **This paper presents a framework for obtaining an optimal policy in model-free Partially Observable Markov Decision Problems (POMDPs) using a recurrent neural network (RNN). A Q-function approximation approach is taken, utilizing a novel RNN architecture with computation and storage requirements that are dramatically reduced when compared to existing schemes. A scalable online training algorithm, derived from the real-time recurrent learning (RTRL) algorithm, is employed. Moreover, stochastic meta-descent (SMD), an adaptive step size scheme for stochastic gradient-descent problems, is utilized as means of incorporating curvature information to accelerate the learning process. We consider case studies of POMDPs where state information is not directly available to the agent. Particularly, we investigate scenarios in which the agent receives indentical observations for multiple states, thereby relying on temporal dependencies captured by the RNN to obtain the optimal policy. Simulation results illustrate the effectiveness of the approach along with substantial improvement in convergence rate when compared to existing schemes.**

Index Terms— **Recurrent neural networks, real-time recurrent learning (RTRL), constraint optimization.**

I. Introduction

Partially Observable Markov Decision Processes (POMDPs) characterize a broad range of real-world problems in which an agent interacts with its environment without being provided with an explicit state representation. In many practical scenarios identical observations may be provided for different states, thereby requiring the agent to rely on memory to infer its state. An agent in a path-searching problem (e.g. maze maneuvering) may receive identical observations for several different positions (or states). In these cases, the agent must recall recent steps in order to infer its precise position. Many problems of interest can be formulated as POMDPs, yet the lack of efficient algorithms results in the limited use of POMDPs in practice. In particular, scalability has been a key issue in obtaining optimal as well as good sub-optimal solutions. Recurrent neural networks (RNNs) are widely acknowledged as an effective tool that can be employed by a wide range of applications that store and process temporal sequences [2][11][6]. This capability makes them particularly attractive as memory-based non-linear function approximation tools for solving POMDPs [9][5]. However, the computational complexity and storage requirements typically associated with RNN realizations have thus far limited their use.

The ability of RNNs to capture complex, nonlinear system dynamics has served as a driving motivation for their study. Most of the proposed RNN learning algorithms rely on the calculation of error gradients with respect to the network parameters, or weights. RNNs are distinguished from static, or feedforward networks, because their gradients are time-dependent (or dynamic). This implies that the current error gradient depends not only on the current input, output and targets, but also on a possibly infinite past. Effectively training RNNs remains a challenging and active research topic.

The learning problem consists of adjusting the weight of the network such that the trajectories have certain specified properties. Perhaps the most common online learning algorithm proposed for RNNs is the Real-Time Recurrent Learning (RTRL) [19][20][3], which calculates gradients in real-time. The gradients at time k are obtained in terms of those at time instant $k-1$. Once the gradients are evaluated, weight updates can be calculated in a straightforward manner. The RTRL algorithm is very attractive in that it is applicable to real-time systems. However, the two main drawbacks of RTRL are the large computational complexity of $O(N^4)$ and, even more critical, the storage requirements of $O(N^3)$, where N denotes the number of neurons in the network.

Many methods have been proposed to reduce the computational complexity of RTRL, such as those utilizing hybrid backpropagation through time (BPTT)/RTRL schemes [13] and others using Green's function [17]. In [20], the sensitivity set for each neuron is reduced to a subgroup of neurons, thereby decomposing the network into several non-overlapping sub-networks. The key advantage of subgrouping in this manner is the immediate reduction in computations to $O(N^4/g^3)$, where g denotes the number of groups. However, for a small number of subgroups, the advantage becomes negligible. If g is large, there is little crossover of training information from different groups, thereby significantly reducing the network's capabilities. The arbitrary selection of subgroups also appears somewhat weak.

To address this concern, recent work has suggested dynamically partitioning the groups with gradient information that is calculated online [3]. Although it constitutes a more intelligent and data-dependent approach, this method is not scalable due to the complex process of dynamically redefining the subgroup boundaries. Moreover, the key problems associated with the number of groups created in [20] remain. Another fundamental limitation of standard RTRL is that it is based on fixed step size gradient descent, i.e. the weight update rule is generally given by

$$w_{ij}(t+1) = w_{ij}(t) - \alpha \frac{\partial J(t)}{\partial w_{ij}}, \tag{1}$$

where $J(t)$ denotes the error function at time t and w is the weight (parameter) space. This often results in extremely slow convergence rates. However, most techniques proposed for accelerating the learning process rely on real-time computation of the Hessian (second derivative of the error function). This incurs extensive storage and computational effort that precludes RTRL based schemes from becoming applicable to large-scale networks (i.e. networks with thousands of nodes and above).

This paper focuses on an alternative approach to reducing the resource requirements of online RNN learning as well as improving its convergence properties, particularly in the context of obtaining policy in POMDPs. This paper therefor addresses three key topics. First, it proposes a reduction of the sensitivities of each neuron to weights associated with its incoming and outgoing connections. This approach results in a localized architecture and learning algorithm that lends itself to hardware realization, while retaining the core desirable capabilities of RTRL. Second, an adaptive step size algorithm, based on stochastic meta-descent (SMD), is derived, thereby substantially improving the learning process at a computational complexity that is comparable to that of regular gradient descent. Third, a POMDP framework is presented and evaluated, employing the proposed RNN architecture. It is shown that the proposed framework offers improved performance in terms of both accuracy as well as convergence rate when compared to existing schemes.

The rest of this paper is structured as follows. Section II provides a brief overview of the RTRL algorithm. In Section III, the TRTRL algorithm is described and analyzed. Section IV develops stochastic meta-descent for TRTRL, while in Section VI experimental results for POMDP case studies are presented. Finally, in Section VII the conclusions are drawn.

II. Overview of RTRL

In this section we briefly describe the RTRL algorithm. Let us assume that a network consists of a set of N fully connected neurons and a set of M inputs. Further, $T \in N$ will denote the set of neurons for which there is a target. Let $w_{ij}(t)$ denote the weight (i.e. the synaptic strength) associated with the link originating from neuron j towards neuron i at time t. The net input to neuron k, $s_k(t)$, is defined as the weighted sum of all activations in the network, $z_l(t)$. Based on standard RTRL terminology, we define the activation function of node k at time $t+1$ to be

$$y_k(t+1) = f_k\left(s_k\left(t\right)\right), \tag{2}$$

where

$$s_k(t) = \sum_{l \in N \cup M} w_{kl} z_l(t), \tag{3}$$

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in M \\ y_k(t) & \text{if } k \in N \end{cases} \tag{4}$$

and the non-linear activation function, $f(\cdot)$, maps $s_k(t)$ to the range [0,1]. The overall network error at time t is defined by

$$J(t) = \frac{1}{2}\sum_{k \in T}\left[d_k(t) - y_k(t)\right]^2 \tag{5}$$

$$= \frac{1}{2}\sum_{k \in T}\left[e_k(t)\right]^2 \tag{6}$$

where $d_k(t)$ denotes the desired target value for output k at time t. Correspondingly, the error is minimized along a negative multiple of the performance measure gradient. The online calculation of the gradients is achieved by exploiting the following relationship:

$$-\frac{\partial J(t)}{\partial w_{ij}(t)} = \sum_{k \in T} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}}. \tag{7}$$

By identifying the partial derivatives of the activation functions with respect to the weights as sensitivity elements, and denoting the notation by

$$p^k_{ij}(t) = \frac{\partial y_k(t)}{\partial w_{ij}}, \tag{8}$$

we obtain the following recursive equation:

$$p^k_{ij}(t+1) = f'_k\left(s_k\left(t\right)\right)\left[\sum_{l \in N} w_{kl} p^l_{ij}(t) + \delta_{ik} z_j(t)\right], \tag{9}$$

where $p^k_{ij}(0) = 0$ and δ_{ik} is the Kronecker delta defined as

$$\delta_{ij}(t) = -\frac{\partial J(t)}{\partial w_{ij}}. \tag{10}$$

Equations (9) and (10) allow one to obtain the performance gradient at any given time.

As can be seen from these equations, each neuron is required to perform $O(N^3)$ multiplications yielding an overall complexity of $O(N^4)$. Moreover, the storage requirements are dominated by the weights $O(N^2)$ and, more importantly, the sensitivity matrices containing $p^k_{ij}(t)$, which are $O(N^3)$. Due to the distributed nature of the network, the calculation can be reduced significantly by having each neuron compute its sensitivities in parallel. If performed in hardware, these computation processes can be accelerated by exploiting pipelining and module replication. However, unlike the computational requirements, the storage requirements cannot be reduced as they constitute a crucial component in the weight update procedure.

Several schemes that have been presented in the literature aim to reduce the storage complexity associated with RTRL. A unifying theme of these methods comprises of subgrouping the neurons into multiple, non-overlapping subnetworks. Although the computational gain is significant, the storage requirements remain high, in particular when a small set of subgroups is employed. We next describe the main contribution of this paper, which focuses on an alternative method for reducing the resource requirements in RTRL.

III. Truncated Real-Time Recurrent Learning (TRTRL)

TRTRL is a variation on RTRL, first introduced in [1], which was designed to overcome the scalability limitations of RTRL while retaining its key performance attributes. TRTRL accomplishes this goal by reducing the amount of information that each neuron is required to consider as it performs its computations. Let us begin with several key definitions that will guide us through the discussion:

Definition 1: Let I_j denote the set of nodes that have a direct link (and, hence, a unique associated weight) to node j. We shall refer to this set as the *ingress* set of node j.

Definition 2: Let E_j denote the set of nodes that node j has a link (and, hence, a unique associated weight) to. We shall refer to this set as the *egress* set of node j.

It should be noted that a node can reside within both ingress and egress sets of another node. Moreover, for the purpose of notation convenience, we shall consider the feedback (i.e. recurrent) link that each node has to itself, to be part of the node's egress set. Consequently, the basic assumption in TRTRL is that the sensitivities of each neuron are limited to its ingress and egress set. This means that, coarsely speaking, a neuron's activation is not directly sensitive to any weight that is not in the neurons ingress or egress set. The only exception to this rule pertains to neurons with targets, as will be elaborated on later.

By localizing the information processed by each neuron, the calculation of equation 9 comprises of three main components. First, its ingress sensitivity function is given by

$$p^i_{ij}(t+1) = f'_i\left(s_i\left(t\right)\right)\left[w_{ij}p^j_{ij}(t) + z_j(t)\right] \ \ \forall i \notin T, i \neq j \tag{11}$$

Notice that the summation from equation (9) is reduced to a single multiplication since $p^l_{ij} = 0$ for all $l \neq i$. Second, following a similar rationale to that applied to the ingress set, the sensitivities pertaining to the egress set of node i are given by

$$p^i_{ji}(t+1) = f'_i\left(s_i\left(t\right)\right)\left[w_{ij}p^j_{ji}(t) + \delta_{ji}y_i(t)\right] \ \ \forall i \notin T \tag{12}$$

From the above two expressions it becomes evident that the aggregate computational load for each neuron is $O(N)$ (in fact, rather close to $2N$).

In order to complete the description of TRTRL, an update rule must be derived for the output neurons (i.e. neurons with target), for which we once again refer to (1) and (7). Here, there are two possible scenarios. The first is one in which the $T \ll N$, i.e. the number of neurons with targets is significantly smaller than the total number of neurons in the network. In that case, it is expected that the majority of the information will be represented by weights and signals associated with the non-output neurons, in which we make the assumption that the output neurons do not have connecting weights, i.e. $w(i,j) = 0 \ \forall i,j \in T$. For the output neurons, a non-zero sensitivity element must exist in order to provide gradient information required by the weight update rule (see (7)). To comply with this requirement, a direct link is added from each output neuron to each of the N neurons in the network. Consequently, each output neuron, $o \in T$, performs a sensitivity update for each weight in the network. This can be achieved using the following update rule, which applies to all $i \neq o, i \notin T$,

$$p^o_{ij}(t+1) = f'_o\left(s_o\left(t\right)\right)\left[w_{oi}p^i_{ij}(t) + w_{oj}p^j_{ij}(t) + \delta_{io}z_j(t)\right], \tag{13}$$

all other sensitivity elements are null. Following such an update rule has the advantages of keeping computations to a minimum, while high information content is retained due to the structure of the network.

Alternatively, for networks in which $T \longrightarrow N$ (i.e. the number of neurons with targets is almost the same as the number of neurons in the network), (13) suggests that very few of the weights will be non-zero. This gives rise to the need for a revised formulation of (13). If weights between output neurons (and themselves) are non-zero, the respective update rule for $i \neq o$ and $j \neq o$ becomes:

$$\begin{aligned} p^o_{ij}(t+1) \ &= \ f'_o\left(s_o\left(t\right)\right)[w_{oi}p^i_{ij}(t) + w_{oj}p^j_{ij}(t) + \\ &\sum_{l \in T} w_{ol}p^l_{ij}(t) + \delta_{io}z_j(t)], \end{aligned} \tag{14}$$

while for all other cases,

$$p^o_{ij}(t+1) = f'_o\left(s_o\left(t\right)\right)\left[\sum_{l \in T} w_{ol}p^l_{ij}(t) + \delta_{io}z_j(t)\right]. \tag{15}$$

The partial sensitivity matrix is invariant to the fact that there may still exist a unique weight between any two neurons in the network. The full calculation of equation (9) must be performed for all input, bias and output units. The only difference between TRTRL and RTRL, in this context, is that neurons are limited in the sensitivities. To that end, TRTRL is highly localized since neurons are no longer required to fetch information that may be located at a remote part of the network. If implemented in hardware, localizing the memory access is key to guaranteeing high-speed of execution. It should be noted that this formalism yields an overall computational complexity of $O(KN^2)$, where $K = |T|$ denotes the number of output neurons in the network. Moreover, storage complexity is $O(N^2)$.

IV. Overview of Stochastic Meta-Descent

A. Background and Motivation

The objective of the TRTRL algorithm, which is essentially an online optimization technique, is to minimize a global error function, J, such that the network's future outputs will be closer to their designated targets. What makes TRTRL and its variants unique is that they are online schemes, whereby each time step an error is provided that is reflected in network parameters (i.e. weights) updates. As such, TRTRL is a *stochastic gradient* based method that aims to optimize the network's performance by utilizing instantaneous gradient information. Network weights are updated iteratively along the negative gradient direction,

$$w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_{ij}(t), \tag{16}$$

where α is the learning rate parameter. In practice, the learning rate α is set to a small constant value in order to guarantee convergence of the training algorithm and avoid oscillations at instances where the error function is steep. However, this approach considerably slows down training since, in general, a small learning rate may not be appropriate for all portions of the error surface[4]. To address this issue, stochastic meta-descent (SMD) [15] [16] applies an adjustable learning rate for every connection (weight) in the network, in an attempt to use not only the gradient but also the second derivative of the error function as means of accelerating the learning process.

B. Stochastic Meta-Descent

In this section, we briefly describe the SMD algorithm[15]. As an alternative to utilizing small, identical constant learning rates for all network weight updates, SMD employs an independent learning rate for each weight. Accordingly, the weight update rule is given by

$$w_{ij}(t+1) = w_{ij}(t) + \lambda_{ij}(t)\delta_{ij}(t), \tag{17}$$

where $\lambda_{ij}(t)$ is the learning rate for weight w_{ij} at time t. Moreover, the local learning rates are independently adapted by exponentiated gradient descent. In this way, they can cover a wide dynamic range while remaining strictly positive[7] [8]. Accordingly, the following learning rate update rule is used:

$$\ln \lambda_{ij}(t) = \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial \ln \lambda_{ij}}, \tag{18}$$

where μ is a global meta-learning rate. Using the chain rule, the above can be rewritten as

$$\begin{aligned} \ln \lambda_{ij}(t) &= \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial w_{ij}(t)} \frac{\partial w_{ij}(t)}{\partial \ln \lambda_{ij}} \\ &= \ln \lambda_{ij}(t-1) + \mu \delta_{ij}(t) v_{ij}(t) \end{aligned} \tag{19}$$

where

$$v_{ij}(t) = \frac{\partial w_{ij}(t)}{\partial \ln \lambda_{ij}}. \tag{20}$$

This approach rests on the assumption that each element of λ affects J only through the corresponding element of w. To avoid an expensive exponentiation for each weight update, (19) is further simplified by exploiting the linearization $e^{\mu} = 1 + \mu$, valid for small $|\mu|$, to yield

$$\lambda_{ij}(t) = \lambda_{ij}(t-1) \max\left(\rho, 1 + \mu \delta_{ij}(t) v_{ij}(t)\right), \tag{21}$$

where ρ (typically around 0.5) is a safeguard factor against unreasonably small, or negative, values. Meta-level gradient descent remains stable as long as $\delta_{ij}(t)v_{ij}(t), \forall\, i,j$ does not stray away from unity. Next, v_{ij} is expressed as a gradient trace that measures the long-term impact of a change in a local learning rate to its corresponding weight. Accordingly, the SMD algorithm defines v_{ij} as an exponential average of the effect of all past learning rates on the new weight values, such that

$$v_{ij}(t+1) = \sum_{k=0}^{\infty} \beta^k \frac{\partial w_{ij}(t+1)}{\partial \ln \lambda_{ij}(t-k)}, \tag{22}$$

where the coefficient $0 < \beta < 1$ determines the time scale over which long-term dependencies are taken into account. (22) can be effectively approximated to yield the following:

$$v_{ij}(t+1) = \beta v_{ij}(t) + \lambda_{ij}(t) \left(\delta_{ij}(t) - \beta \left(H_t v(t)\right)_{ij} \right), \tag{23}$$

where $v_{ij}(0) = 0, \forall i,j$ and H_t denotes the instantaneous Hessian (the matrix of second derivatives $\partial^2 J / \partial w_{ij} w_{kl}$ of the error J with respect to each pair of weights) at time t. The two equations (21) and (23) complete the updating of the learning rates λ_{ij} for each w_{ij}.

C. SMD for TRTRL

In applying SMD to TRTRL, the primary task is to derive an efficient algorithm for obtaining $H_t v_t$. At first glance, this might suggest a computationally heavy process. Fortunately this is not the case, since there are very efficient indirect methods for computing the product of the Hessian with an arbitrary vector [12] [10]. To prevent negative eigenvalues from causing (23) to diverge, SMD uses an extended Gauss-Newton approximation that also admits a fast matrix-vector product. Pearlmutter [12] presented an exact and numerically stable procedure to compute $H_t v_t$ with a computational complexity of $O(n)$ and no need to explicitly calculate or store the matrix H_t. We begin by reviewing this technique. It has been shown that the product of a Hessian H with any arbitrary vector, v, can be computed as

$$Hv = R_v\{\nabla_w\} = \frac{\partial}{\partial r} \nabla_{(w+rv)} \,|_{r=0} \tag{24}$$

where $R_v\{\cdot\}$ is a differential operator and r is a real value. ∇_w is the gradient of the optimized function with respect to the adjustable parameters w. rv is considered a small perturbation to ∇_w in the direction of v. In the context of neural networks, ∇_w is the gradient of the error function to the weights and v is the gradient trace defined in (20). Applying the $R_v\{\cdot\}$ operator to TRTRL, we obtain

$$\begin{aligned} -\left(H_t v(t)\right)_{ij} &= R_v\left\{-\nabla_{w_{ij}}\right\} \\ &= R_v\left\{\delta_{ij}(t)\right\} \\ &= R_v\left\{-\frac{\partial J(t)}{\partial w_{ij}}\right\} \\ &= R_v\left\{\sum_{o \epsilon output} e_o(t) p_{ij}^o(t)\right\} \end{aligned} \tag{25}$$

$$\begin{aligned} &= \sum_{o \epsilon output} \left[e_o(t) R_v\left\{p_{ij}^o(t)\right\} + \right. \\ &\qquad \left. R_v\left\{e_o(t)\right\} p_{ij}^o(t) \right] \\ &= \sum_{o \epsilon output} \left[e_o(t) R_v\left\{p_{ij}^o(t)\right\} - \right. \\ &\qquad \left. R_v\left\{y_o(t)\right\} p_{ij}^o(t) \right] \end{aligned}$$

Next, we need to calculate $R_v\{y_o(t)\}$ and $R_v\{p_{ij}^o(t)\}$. From equation (3), we note that

$$R_v\{s_o(t)\} = \sum_{l \epsilon U \cup I} v_{ol}(t) z_l(t), \tag{26}$$

such that from (2),

$$R_v\left\{y_o(t)\right\} = f'(s_o(t))R_v\left\{s_o(t)\right\}. \tag{27}$$

By the same token and (13),

$$\begin{aligned} R_v\left\{p^o_{ij}(t)\right\} &= f''\left(s_o(t)\right)R_v\left\{s_o(t)\right\} \\ &\cdot\left[w_{oi}p^i_{ij}(t) + w_{oj}p^j_{ij}(t) + \delta_{io}z_j(t)\right] \\ &+ f'(s_o(t))\left[v_{oi}p^i_{ij}(t) + v_{oj}p^j_{ij}(t)\right] \end{aligned} \tag{28}$$

Note that the computation of $H_t v_t$ only incurs calculations at the output neurons, thus adding little to the overall computations. It should also be noted that the calculation of $H_t v_t$ can be thought of as a concurrent and adjoint process to the gradient calculation, with a similar computational complexity of $O(KN^2)$. Moreover, the storage requirements remain $O(N^2)$. For a linear transfer function at the output neurons (i.e. $f\left(s_o(t)\right) = s_o(t)$), we have $f'\left(s_o(t)\right) = 1$ and $f''\left(s_o(t)\right) = 0$, resulting in the simplified expression:

$$R_v\left\{p^o_{ij}(t)\right\} = v_{oi}p^i_{ij}(t) + v_{oj}p^j_{ij}(t). \tag{29}$$

D. Adaptation of the Global Meta-learning Rate μ

The original SMD technique does not consider any adaptation of the global meta-learning rate parameter, μ. In fact, the latter is often viewed as the "learning rate of the learning rate", with typical values in the order of 0.1. To ensure faster convergence and stability of the algorithm as a whole, we introduce an adaptive global meta-learning rate by the same heuristic techniques of SuperSAB [10] [18]. We increase the value of μ if a positive correlation between successive gradients of the error function with respect to learning rate is observed, otherwise μ is decreased. Let φ be the negative gradient of the error function with respect to the exponentiated learning rate such that

$$\varphi_{ij}(t) = -\frac{\partial J(t)}{\partial \ln \lambda_{ij}} = \delta_{ij}(t)v_{ij}(t). \tag{30}$$

Accordingly, $\mu_{ij}(t)$ is updated in the following manner:

$$\mu_{ij}(t) = \mu_{ij}(t-1)\left(1 + \eta\varphi_{ij}(t)\varphi_{ij}(t-1)\right), \tag{31}$$

where $\eta = .05$ is a small positive constant. Moreover, μ_{ij} is bounded by $[\mu_{\min} = 0.01, \mu_{\max} = 5]$ in order to ensure stability and smoother learning.

Although SMD proves to yield a generally stable learning process, occasional divergence in weight values can occur. A heuristic that has been found to eliminate such instabilities is to simply limit both the weights and the weight changes. The latter are restricted to the range $[-0.125, 0.125]$, while the former to $[-4, 4]$. This has shown to carry minimal learning rate degradation, while practically guaranteeing stability.

E. Discussion on Storage and Computational Complexity

Primary benefits of TRTRL, from an implementation perspective, are the substantial reductions in computation complexity and storage requirements. Computation time is dominated by the calculation of the sensitivity elements. While in the original RTRL scheme, each neuron is required to perform $O(N^3)$ floating-point operations (flops), TRTRL requires only $O(N)$. Note that SMD necessitates approximately three times the flops involved in regular gradient computations. This results in an overall (network-level) computational complexity of $O(N^2)$, instead of $O(N^4)$ that characterizes RTRL.

A similar reduction in resources is observed in the storage requirements of TRTRL. All N^3 elements of the sensitivity matrix are required in RTRL, while TRTRL only operates on $2N$ sensitivities per neuron. As such, the overall storage requirement drops from $O(N^3)$ to $O(N^2)$. It should be noted that, as opposed to RTRL, TRTRL is a highly localized algorithm. This contributes to the more effective implementation prospect of the scheme in hardware. Moreover, it is interesting to note that although this paper addresses the case of fully-connected networks, the TRTRL formalism is not restricted to such cases. In fact, assuming that each node is only connected to M other nodes, the computational complexity becomes $O(KMN)$ while storage is reduced to $O(MN)$. The only constraint imposed in such cases is that each node has a direct link to the output neurons (as means of propagating error information), as dictated by (7).

V. Experimental Results

A. Direct-Policy Approximate Dynamic Programming with Softmax Action Selection

We introduce `RNNPOMDP`, an online stochastic gradient learning control framework, which utilizes a recurrent neural network for Q-function approximation. The controller is constructed of a fully-connected RNN with one output neuron that predicts the state-action value, based on which the softmax algorithm [14] is used to determine the actions. The goal is to learn a stochastic control scheme that yields a near-optimal policy. All the RNN nodes use a sigmoid activation function with the exception of the output node which has a linear activation function. We apply Q-learning and use the TRTRL-SMD algorithm to train the RNN. The RNN is trained with reference to the temporal difference error,

$$\delta_t = r_{t+1} + \gamma \max_i\{Q(\vartheta_{t+1}, a_i)\} - Q(\vartheta_t, a_t), \tag{32}$$

where ϑ is the observation, a_i are within the set of possible actions, r denotes the single-step reward, and γ is the discounting factor (set to 0.8). For each step, the RNN is used to evaluate the Q-function for all possible actions, followed by softmax action selection. Since the neural network has state (by means of activations), previous activations and weights are stored and updated during the subsequent time step, upon evaluation of the temporal difference error. The algorithm is given in Table 1.

B. Simple Three-State POMDP

We first consider a simple 3-state POMDP, as used by Baxter et al. and Schraudolph et al. [14]. The RNN consisted of 5 internal neurons and one output neuron. The observation at each state is a vector denoted by (o_1, o_2). Of the two possible transitions from each state, the preferred one occurs

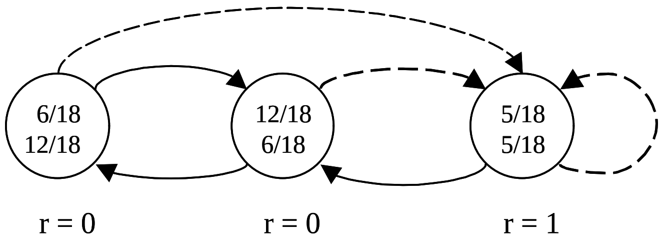

Fig. 1. Baxter et al.'s simple 3-state POMDP. States are labelled with their observable feature vectors and instantaneous reward r; arrows indicate the 80% likely transition for the first (solid) and second (dashed) action.

with 80% probability, while the other with 20%, as illustrated in Fig (1). The preferred transition is determined by the action of a simple probabilistic adaptive controller that receives two state-dependent feature values as input, and is trained to maximize the expected average reward.

```
RNN $Q-$learning with softmax action selection
1. Given:
(a) an ergodic POMDP with observation space $\Theta$,
    action space $A$, and bounded reward;
(b) an RNN with initial weights $w_0$ and function
    mapping observation-action pair to real value
    $f : \Theta \times A \rightarrow R$
2. For $t = 1$ to $\infty$ :
(a) Interact with POMDP:
    1) observe $\vartheta_t \in \Theta$, evaluate all actions $a_i \in A(\vartheta_t)$
       via the RNN with $Q(\vartheta_t, a_i) = f(\theta_t, a_i, w_t)$;
    2) calculate the probability of taking each action in $\vartheta_t$
       using softmax: $\Pr\{a_i \text{ in } \vartheta_t\} = \frac{e^{Q(\vartheta_t,a_i)/\tau}}{\sum_i e^{Q(\vartheta_t,a_i)/\tau}}$;
    3) select action based on probability and observe
       the reward $r_{t+1}$;
(b) Update the activations (i.e. internal states) and
    weights of the RNN:
    1) input the previous observation-action pair
       $(\vartheta_t, a_t)$ to RNN;
    2) update prior time step weights and sensitivities,
       based on $(\vartheta_t, a_t)$, and (17), with
       temporal difference error given by
         $\delta_t = r_{t+1} + \max_i\{Q(\vartheta_{t+1}, a_i)\} - Q(\vartheta_t, a_t)$;
```

Table 1: Q-function approximation based POMDP learning using the TRTRL-SMD algorithm

In this test, the free parameter for softmax algorithm was set to $\tau = 0.5$; for TRTRL-SMD, the parameters were configured to the following initial values: $\lambda_{ij}(0) = 0.01, \forall$ i, j, $\rho = 0.5, \beta = 0.95, \mu_{ij}(0) = 0.1$, $\mu_{ij} \in [0.01, 0.5]$, and $\eta = 0.05$. We collected data from 500 independent runs with random seeds and initial conditions, and compared the convergence rates with the ones obtained for SMDPOMDP [14], which is a feed-forward neural network based approach. The comparison is shown in Fig 2. Both algorithms converge asymptotically to the optimal average reward ($R = 0.8$), with the RNNPOMDP algorithm converging faster in terms of process steps.

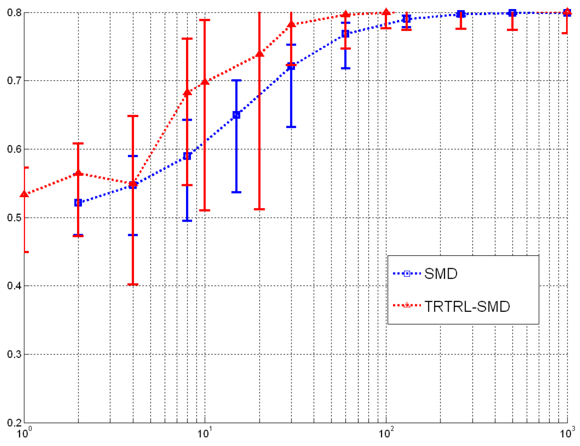

Fig. 2. Comparison of SMDPOMDP and RNNPOMDP applied to a simple 3-state POMDP

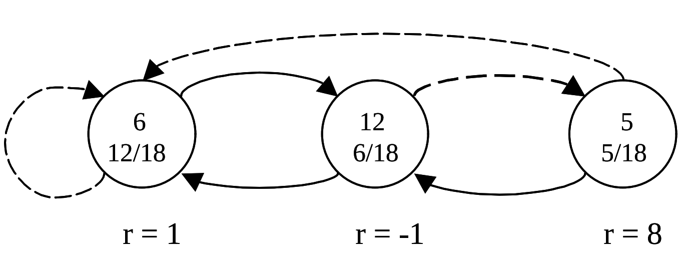

Fig. 3. Schraudolph et al.'s modified 3-state POMDP

C. *Modifed Three-State POMDP*

The first test was a simple three-state POMDP with the property that greedy maximization of instantaneous reward leads to the optimal policy. Schraudolph et al. [14] introduced a more challenging problem which assigns a deceptive instantaneous reward to a transition state. In the modified POMDP 3, the highest reward state can only be reached through an intermediate state with a negative reward. The state features (observations) were also modified so as to create an ill-conditioned input to the controller. For this test, we used a 10-neuron RNN with the same settings as stated above and trained the network according to the algorithm described in table 1. Fig 4 illustrates that while SMDPOMDP reaches the optimal performance after approximately 10^6 iterations, RNNPOMDP converged to the optimal average reward almost 10 times faster. Moreover, the RNN trained with SMD also yielded considerable performance improvement prior to converging to the optimal average reward.. This further supports the notion that utilizing RNNs in this context results in better approximation of long-term rewards.

D. *Four-State POMDP*

While the translation of features to inferred states was indirect in the above test cases, it did not constitute a true POMDP in the sense that observation-to-state is ambiguous. We therefore studied a four-state POMDP, as depicted in

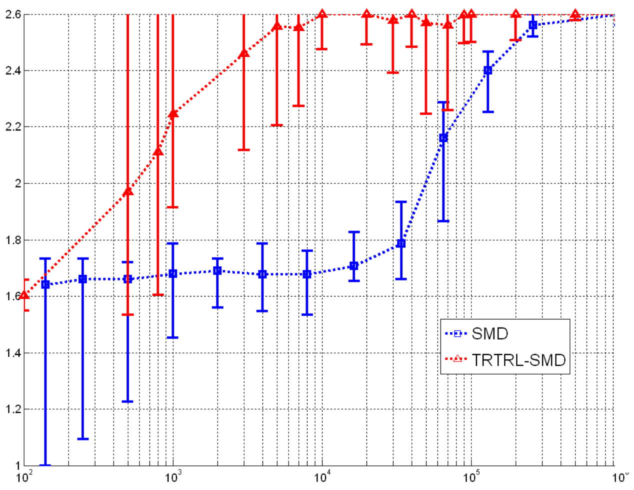

Fig. 4. Comparison of SMDPOMDP and RNNPOMDP applied to the modified 3-state POMDP

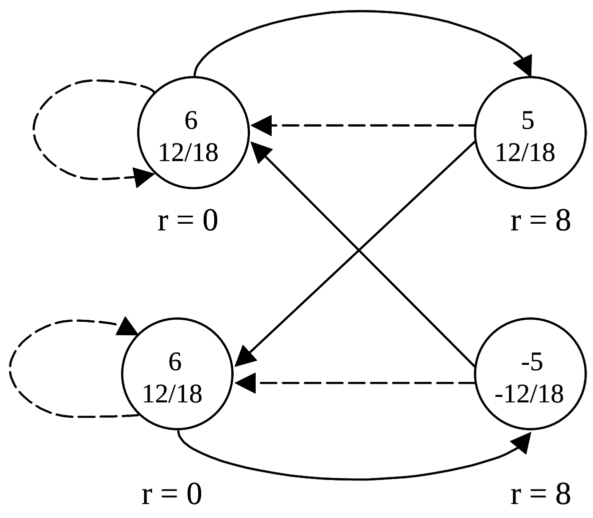

Fig. 5. 4-state POMDP with identical observations for different states.

Fig 5 . The two states on the left-hand side have the same observable features but different preferred actions. When the agent visits any of the two states on the right-hand side, it transitions to the states on the left-hand side (as depicted in the figure), regardless of the action taken. The observation is memory-dependent in the sense that each state has a distinct preceding observation. The controller needs to memorize preceding observations in order to determine the optimal action for each of the two states. Hence, stateful function approximation achieved by the RNN is mandatory. The RNN in this case consisted of 15-neurons with the same initial setup as stated above. Fig 6 demonstrates the asymptotic convergence to the policy. It is worth mentioning that a near-optimal policy is obtained after only 10^4 iterations.

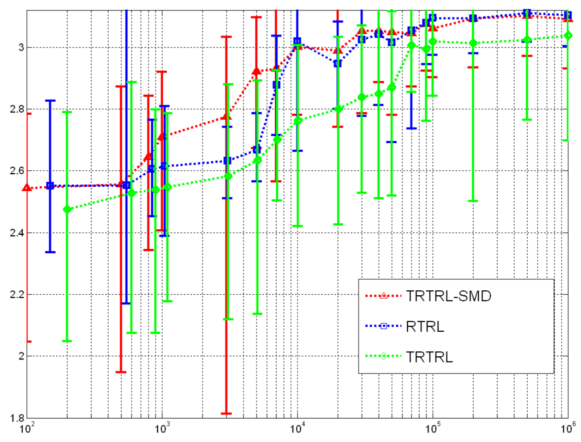

Fig. 6. Comparison of RTRL, TRTRL and TRTRL-SMD applied to the 4-state POMDP

VI. Conclusions

In this paper, we presented a recurrent neural network based Q-learning POMDP framework. An efficient realization of the RNN yielded a scalable architecture, while training was improved via the stochastic-meta descent technique. Simulation results of several POMDP test cases clearly demonstrated the performance advantages of the proposed scheme, with respect to both accuracy in estimating the average reward as well as speed and precision in convergence.

VII. Acknowledgements

The authors would like to thank Derek Rose for his review of the manuscript. This work has been partially supported by the Department of Energy research contract DE-FG02-04ER25607.

References

[1] D. Budik and I. Elhanany, “TRTRL: a localized resource-efficient learning algorithm for recurrent neural networks,” in *IEEE Midwest Symposium on Circuits and Systems (MWSCAS)*, August 2006, puerto Rica.

[2] A. Delgado, C. Kambhampati, and K. Warwick, “Dynamic recurrent neural network for system identification and control,” in *IEE Proceedings - Control Theory and Applications*.

[3] N. Euliano and J. Principe, “Dynamic subgrouping in RTRL provides a faster o(n^2) algorithm,” in *Acoustics, Speech, and Signal Processing, 2000. ICASSP ’00. Proceedings. 2000 IEEE International Conference on*, vol. 6, Istanbul, 2000, pp. 3418–3421.

[4] S. E. Fahlman, “An empirical study of learning speed in back-propagation networks,” *Computer Science Technical Report*, 1988.

[5] F. J. Gomez and J. Schmidhuber, “Co-evolving recurrent neurons learn deep memory pomdps,” in *GECCO ’05: Proceedings of the 2005 conference on Genetic and evolutionary computation*. New York, NY, USA: ACM Press, 2005, pp. 491–498.

[6] L. Jain, *Recurrent Neural Networks*. CRC Press, February 2000.

[7] J. Kivinen and M. Warmuth, “Exponentiated gradient versus gradient descent for linear predictors,” *Tech. Rep. UCSC-CRL-94-16*, 1994.

[8] J. Kivinen and M. K. Warmuth, “Additive versus exponentiated gradient updates for linear prediction,” in *Proc. of the twenty-seventh annual ACM symposium on Theory of computing*, 1995, pp. 209–218.

[9] L. Lin and T. Mitchell, “Memory approaches to reinforcement learning in non-Markovian domains,” Pittsburgh, PA, USA, Tech. Rep., 1992.

[10] G. D. Magoulas, M. N. Vrahatis, and G. S. Androulakis, “Improving the convergence of the backpropagation algorithm using learning rate adaptation methods,” *Neural Computation*, vol. 11, no. 7, pp. 1769–1796, 1999.

[11] D. Nguyen and B. Widrow, “Neural networks for self-learning control system,” *IEEE Cont. Sys.*, vol. 10, no. 4, pp. 18–23, April 1990.

[12] B. A. Pearlmutter, “Fast exact multiplication by the Hessian,” *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994.

[13] J. Schmidhuber, “A fixed size storage o(n^3) time complexity learning algorithm for fully recurrent continually running networks,” *Neural Computation*, vol. 4, pp. 243–248, 1992.

[14] N. Schraudolph, J. Yu, and D. Aberdeen, “Fast online policy gradient learning with smd gain vector adaptation,” in *Advances in Neural Information Processing Systems 18*, Y. Weiss, B. Schölkopf, and J. Platt, Eds. Cambridge, MA: MIT Press, 2006, pp. 1185–1192.

[15] N. N. Schraudolph, “Local gain adaptation in stochastic gradient descent,” *Tech. Rep. IDSIA-09-99*, Aug 1999.

[16] N. N. Schraudolph, J. Yu, and D. Aberdeen, “Fast online policy gradient learning with SMD gain vector adaptation,” *19th Annual Conference on Neural Information Processing Systems*, Dec 2005, vancouver, Canada.

[17] G.-Z. Sun, H.-H. Chen, and Y.-C. Lee, “Green’s function method for fast on-line learning algorithm of recurrent neural networks.” in *NIPS*, 1991, pp. 333–340.

[18] T. tollenaere, “Supersab: fast adaptive backpropagation with good scaling properties,” *Neural Networks*, vol. 3, no. 5, pp. 561–573, 1990.

[19] R. J. Williams and D. Zipser, “A learning algorithm for continually running fully recurrent neural networks,” *Neural Computation*, no. 1, pp. 270–280, 1989.

[20] D. Zipser, “A subgrouping strategy that reduces complexity and speeds up learning in recurrent networks,” *Neural Computation*, no. 1, pp. 552–558, 1989.