

## A Fast and Scalable Recurrent Neural Network Based on Stochastic Meta Descent

Zhenzhen Liu and Itamar Elhanany

**Abstract**—This brief presents an efficient and scalable online learning algorithm for recurrent neural networks (RNNs). The approach is based on the real-time recurrent learning (RTRL) algorithm, whereby the sensitivity set of each neuron is reduced to weights associated with either its input or output links. This yields a reduced storage and computational complexity of  $O(N^2)$ . Stochastic meta descent (SMD), an adaptive step size scheme for stochastic gradient-descent problems, is employed as means of incorporating curvature information in order to substantially accelerate the learning process. We also introduce a clustered version of our algorithm to further improve its scalability attributes. Despite the dramatic reduction in resource requirements, it is shown through simulation results that the approach outperforms regular RTRL by almost an order of magnitude. Moreover, the scheme lends itself to parallel hardware realization by virtue of the localized property that is inherent to the learning framework.

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

### I. INTRODUCTION

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. The ability of RNNs to capture complex, nonlinear system dynamics has served as a driving motivation for their study. RNNs have the potential to be effectively used in modeling, system identification, and adaptive control applications [1]–[3], to name a few, where other techniques may fall short. Most of the proposed RNN learning algorithms rely on the calculation of error gradients with respect to the network weights. What distinguishes recurrent neural networks from static, or feedforward networks, is the fact that the gradients are time dependent or dynamic. This implies that the current error gradient does not only depend on the current input, output, and targets, but rather on its possibly infinite past. How to effectively train RNNs remains a challenging and active research topic.

The learning problem consists of adjusting the parameters (weights) 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) [4]–[7], which calculates gradients at time  $k$  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 [8], and others using Green's function [9]. In [5], the sensitivity set for each neuron is reduced to a subgroup of neurons, thereby decomposing the network into several nonoverlapping subnetworks. 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 in accordance with gradient information being calculated online [6]. 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 [5] 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

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

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). The latter 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 more). Moreover, other quasi-Newton methods introduced in the literature, such as BFGS and LBFGS [10], are applicable for offline batch training rather than stochastic gradient applications.

This brief focuses on an alternative approach to reducing the resource requirements of online RNN learning as well as improving its convergence properties. First, it proposes to reduce the sensitivities of each neuron to weights associated with its incoming and outgoing connections. This results in a localized algorithm that lends itself to hardware realization, while retaining the core capabilities associated with RTRL. Second, an adaptive step size algorithm, based on stochastic meta descent (SMD), is introduced, which substantially improves the learning process at a computational complexity that is comparable to that of regular gradient descent. Finally, subgrouping (clustering) is applied in the new algorithm that further improves its scalability and efficiency.

The rest of this brief is structured as follows. Section II provides a brief overview of the RTRL algorithm. In Section III, the truncated real-time recurrent learning (TRTRL) algorithm is described and analyzed. Section IV develops SMD for TRTRL and Section V introduces clustered TRTRL, while in Section VI, simulation results 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. 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  $t$ . Let  $y(t)$  denote the  $n$ -tuple of outputs of the internal neurons and  $x(t)$  denote the  $m$ -tuple of external input to the network at time  $t$ .  $z(t)$  is formed by concatenating  $y(t)$  and  $x(t)$ , with  $U$  denoting the set of indices  $k$  such that  $z_k$  is the output of an internal neuron and  $I$  the set of indices  $k$  for which  $z_k$  is an external input. Further,  $T \subseteq U$  will denote the set of neurons for which there is a target. The net input to neuron  $k$ ,  $s_k(t)$ , is defined as the weighted sum of all activations in the network  $z(t)$ . Based on standard RTRL terminology, we define the activation function of node  $k$  at time  $t + 1$  to be

$$y_k(t+1) = f_k(s_k(t)) \quad (2)$$

Manuscript received June 21, 2007; revised November 28, 2007; accepted January 24, 2008. This work was supported in part by the Department of Energy under Research Contract DE-FG02-04ER25607.

The authors are with the Electrical Engineering and Computer Science, The University of Tennessee, Knoxville, TN 37996 USA (e-mail: itamar@ieee.org).

Digital Object Identifier 10.1109/TNN.2008.2000838

where

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

$$z_k(t) = \begin{cases} x_k(t), & \text{if } k \in I \\ y_k(t), & \text{if } k \in U \end{cases} \quad (4)$$

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

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

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

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:

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

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

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \quad (8)$$

we obtain the following recursive equation:

$$p_{ij}^k(t+1) = f'_k(s_k(t)) \left[ \sum_{l \in N} w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t) \right] \quad (9)$$

where  $p_{ij}^k(0) = 0$  and  $\delta_{ik}$  is the Kronecker delta. Equations (9) and (14) 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  $p_{ij}^k(t)$ , 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 is subgrouping the neurons into multiple, nonoverlapping 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 brief, which focuses on an alternative method for reducing the resource requirements in RTRL.

### III. TRUNCATED REAL-TIME RECURRENT LEARNING

TRTRL is a variation on RTRL, first introduced in [11] and [12], 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 resources required for each neuron. Let us begin with several key definitions that would guide us through the discussion.

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

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

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 will 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 in (12).

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

$$p_{ij}^i(t+1) = f'_i(s_i(t)) \left[ w_{ij} p_{ij}^i(t) + z_j(t) \right] \quad \forall i \notin T, i \neq j. \quad (10)$$

Notice that the summation from (9) is reduced to a single multiplication because  $p_{ij}^l = 0$  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

$$p_{ij}^i(t+1) = f'_i(s_i(t)) \left[ w_{ij} p_{ij}^i(t) + \delta_{ji} y_i(t) \right] \quad \forall i \notin T. \quad (11)$$

From (10) and (11), it becomes evident that the aggregate computational load for each neuron is  $O(N)$  (in fact, rather close to  $2N$ ).

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, we assume that  $|T| \ll N$ , 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 nonoutput 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 nonzero sensitivity element must exist 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,  $\forall o \in T$ , performs a sensitivity update for each weight in the network. This can be achieved using the following update rule:

$$p_{ij}^o(t+1) = f'_o(s_o(t)) \left[ w_{oi} p_{ij}^i(t) + w_{oj} p_{ij}^j(t) + \delta_{io} z_j(t) \right]. \quad (12)$$

The advantages of following such an update rule are that computations are kept to a minimum, while high information content is retained due to the structure of the network. 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 because 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, based on which the network parameters (i.e., weights) are updated. 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

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \delta_{ij}(t) \quad (13)$$

where

$$\delta_{ij}(t) = -\frac{\partial J(t)}{\partial w_{ij}} \quad (14)$$

and  $\alpha$  is the learning rate parameter. In practice, the learning rate  $\alpha$  is set to a small constant value in order to guarantee convergence of the training algorithm and avoid oscillations in a direction 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 [13]. To address this issue, SMD [14], [15] 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 [14]. 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

$$w_{ij}(t+1) = w_{ij}(t) + \lambda_{ij}(t) \delta_{ij}(t) \quad (15)$$

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 [16]. Accordingly, the following learning rate update rule is used:

$$\ln \lambda_{ij}(t) = \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial \ln \lambda_{ij}} \quad (16)$$

where  $\mu$  is a global meta-learning rate. Using the chain rule, (16) can be rewritten as

$$\begin{aligned} \ln \lambda_{ij}(t) &= \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial w_{ij}(t)} \frac{\partial w_{ij}(t)}{\partial \ln \lambda_{ij}} \\ &= \ln \lambda_{ij}(t-1) + \mu \delta_{ij}(t) v_{ij}(t) \end{aligned} \quad (17)$$

where

$$v_{ij}(t) = \frac{\partial w_{ij}(t)}{\partial \ln \lambda_{ij}}. \quad (18)$$

This approach rests on the assumption that each element of  $\lambda$  affects  $J$  only through the corresponding element of  $w$ . To avoid an expensive exponentiation for each weight update, (17) is further simplified by exploiting the linearization  $e^\mu = 1 + \mu$ , valid for small  $|\mu|$ , to yield

$$\lambda_{ij}(t) = \lambda_{ij}(t-1) \max(\rho, 1 + \mu \delta_{ij}(t) v_{ij}(t)) \quad (19)$$

where  $\rho$  (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

$$v_{ij}(t+1) = \sum_{k=0}^{\infty} \beta^k \frac{\partial w_{ij}(t+1)}{\partial \ln \lambda_{ij}(t-k)} \quad (20)$$

where the coefficient  $0 < \beta < 1$  determines the time scale over which long-term dependencies are taken into account. Equation (20) can be effectively approximated to yield the following:

$$v_{ij}(t+1) = \beta v_{ij}(t) + \lambda_{ij}(t) (\delta_{ij}(t) - \beta (H_t v(t))_{ij}) \quad (21)$$

where  $v_{ij}(0) = 0$ ,  $\forall i, j$ , and  $H_t$  denotes the instantaneous Hessian (the matrix of second derivatives  $\partial^2 J / \partial w_{ij} \partial w_{kl}$  of the error  $J$  with respect to each pair of weights) at time  $t$ . The two equations (19) and (21) complete the updating of the learning rates  $\lambda_{ij}$  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, because there are very efficient indirect methods for computing the product of the Hessian with an arbitrary vector [17], [18]. To prevent negative eigenvalues from causing (21) to diverge, SMD uses an extended Gauss-Newton approximation that also admits a fast matrix-vector product. Pearlmutter [17] 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

$$Hv = R_v \{ \nabla_w \} = \frac{\partial}{\partial r} \nabla_{(w+rv)} \Big|_{r=0} \quad (22)$$

where  $R_v \{ \cdot \}$  is a differential operator and  $r$  is a real value.  $\nabla_w$  is the gradient of the optimized function with respect to the adjustable parameters  $w$ .  $rv$  is considered a small perturbation to  $\nabla_w$  in the direction of  $v$ . In the context of SMD,  $\nabla_w$  is the gradient of the error function to the weights and  $v$  is the gradient trace defined in (18). Applying the  $R_v \{ \cdot \}$  operator to TRTRL, we obtain

$$\begin{aligned} -(H_t v(t))_{ij} &= R_v \{ -\nabla_{w_{ij}} \} \\ &= R_v \{ \delta_{ij}(t) \} \\ &= R_v \left\{ -\frac{\partial J(t)}{\partial w_{ij}} \right\} \\ &= R_v \left\{ \sum_{o \in \text{output}} e_o(t) p_{ij}^o(t) \right\} \\ &= \sum_{o \in \text{output}} e_o(t) R_v \{ p_{ij}^o(t) \} + R_v \{ e_o(t) \} p_{ij}^o(t) \\ &= \sum_{o \in \text{output}} e_o(t) R_v \{ p_{ij}^o(t) \} - R_v \{ y_o(t) \} p_{ij}^o(t). \end{aligned} \quad (23)$$

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

$$R_v \{ s_o(t) \} = \sum_{i \in U \cup I} v_{oi}(t) z_i(t) \quad (25)$$

such that from (2)

$$R_v \{ y_o(t) \} = f'(s_o(t)) R_v \{ s_o(t) \}. \quad (26)$$

By the same token and (12)

$$R_v \{p_{ij}^o(t)\} = f''(s_o(t))R_v \{s_o(t)\} \cdot \left[ w_{oi}p_{ij}^i(t) + w_{oj}p_{ij}^j(t) + \delta_{io}z_j(t) \right] + f'(s_o(t)) \left[ v_{oi}p_{ij}^i(t) + v_{oj}p_{ij}^j(t) \right]. \quad (27)$$

Note that the computation of  $H_t v_t$  mainly involves information from the output neurons. 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 are still  $O(N^2)$ . For a linear transfer function at the output neurons (i.e.,  $f(s_o(t)) = s_o(t)$ ), we have  $f'(s_o(t)) = 1$  and  $f''(s_o(t)) = 0$ , resulting in the simplified expression

$$R_v \{p_{ij}^o(t)\} = v_{oi}p_{ij}^i(t) + v_{oj}p_{ij}^j(t). \quad (28)$$

#### D. Adaptation of the Global Meta-Learning Rate $\mu$

The original SMD technique does not consider any adaptation of the global meta-learning rate parameter  $\mu$ . 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 [18], [19]. We increase the value of  $\mu$  if a positive correlation between successive gradients of the error function with respect to learning rate is observed, otherwise  $\mu$  is decreased. Let  $\varphi$  be the negative gradient of the error function with respect to the exponentiated learning rate such that

$$\varphi_{ij}(t) = -\frac{\partial J(t)}{\partial \ln \lambda_{ij}} = \delta_{ij}(t)v_{ij}(t). \quad (29)$$

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

$$\mu_{ij}(t) = \mu_{ij}(t-1)(1 + \eta\varphi_{ij}(t)\varphi_{ij}(t-1)) \quad (30)$$

where  $\eta = 0.05$  is a small positive constant. Moreover,  $\mu_{ij}$  is bounded by  $[\mu_{\min} = 0.01, \mu_{\max} = 5]$  to ensure stability and smoother learning.

#### 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 has 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 the  $O(N^4)$  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 brief 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).

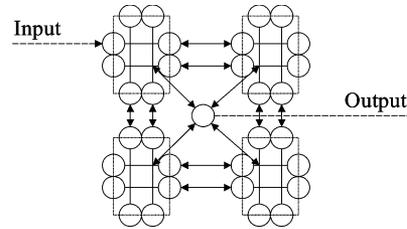


Fig. 1. Diagram of four TRTRL clusters with eight neurons in each. The shared neuron at the center is the output neuron.

## V. CLUSTERED TRTRL

To improve the TRTRL algorithm for hardware scalability, it would be beneficial to further reduce the number of connections between neurons. Following a similar approach to that first discussed in [5], we consider the formation of clusters of neurons, where neurons are connected only to neurons in their cluster. As a result, with the exception of border neurons, sensitivity is restricted to neurons in the same cluster. This implies that each hidden layer node  $i$  only communicates with  $N/B$  other nodes, where  $B$  denotes the number of clusters. The rest of the architecture in this approach remains the same as described previously. In particular, all nodes receive input patterns from the input layer and all nodes have a link to the output layer. It should be noted that there must be some connectivity between clusters, defined as *intercluster connectivity*, to facilitate the propagation of gradient information across the network. Thus, *border neurons* within each cluster are connected to border neurons in other clusters by means of a direct link. A diagram of a clustered structure is illustrated in Fig. 1.

## VI. SIMULATION RESULTS

We performed a comparison between the RTRL, TRTRL, TRTRL with SMD (TRTRL-SMD), and clustered TRTRL algorithms for two commonly employed testbenches that require the network to capture temporal dependencies: frequency doubling and chaotic time-series prediction. In both cases, information that arrives at a given time has strong impact on the value of outputs at subsequent time steps. In essence, it is a measure of the meaningfulness of neuron activations (which are the “soft” state of the network) in order to successfully accomplish the tasks.

### A. Frequency Doubler

The first task chosen was the frequency doubler system. For this task, the network was required to produce a sinusoidal signal that has twice the frequency of the signal applied at its input. The latter is a sinusoid with a 16-sample period while the desired output signal is a sinusoid with an eight-sample period. This is a suitable basic task for the network as the input-to-output mapping is nonlinear and requires memory.

For RTRL, TRTRL, and TRTRL-SMD, the network consisted of a single hidden layer with 15 fully recurrent neurons, one bias input neuron whose (constant) value is 1 and a single linear output neuron. The same set of random initial weights was applied to the three networks each time they were trained. Moreover, an initial learning rate of  $\lambda_{ij}(0) = 0.01, \forall i, j$ , applied to all three algorithms, with TRTRL-SMD gradually adapting its learning rate to accelerate the learning process. The TRTRL-SMD algorithm parameters were configured with the following initial values  $\rho = 0.5, \beta = 0.95, \mu_{ij}(0) = 0.1$ , where  $\mu_{ij} \in [0.01, 5]$ , and  $\eta = 0.05$ .

Fig. 2 depicts the average learning curves for the three algorithms over 50 runs with each sampling point being an average of 100 iterations. While TRTRL converges somewhat slower than RTRL, SMD

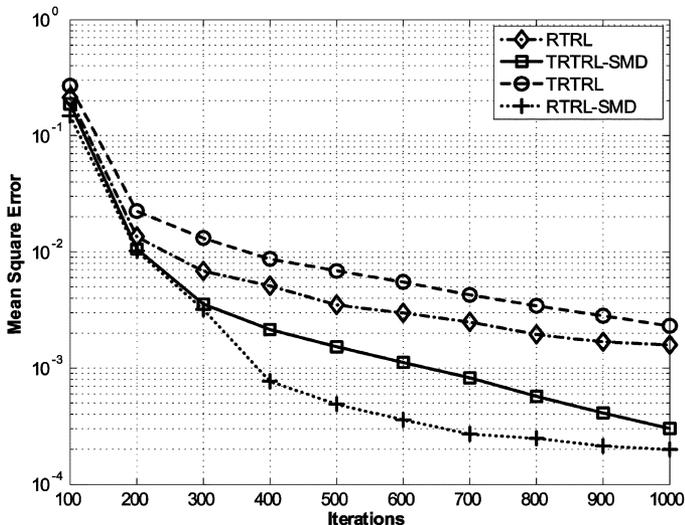


Fig. 2. Average learning curves for the frequency doubler testbench, comparing a 15-neuron fully recurrent network running RTRL, RTRL-SMD, TRTRL, and TRTRL-SMD.

TABLE I  
MSE, STD, AND CONVERGENCE IN CPU TIME FOR RTRL, TRTRL,  
RTRL-SMD, AND TRTRL-SMD LEARNING ALGORITHMS  
IN FREQUENCY DOUBLER TEST

	MSE at Convergence	STD at Convergence	CPU time (sec)
RTRL	0.0016	0.00057	79.97
TRTRL	0.0023	0.00031	18.35
RTRL-SMD	0.0002	0.00014	180.83
TRTRL-SMD	0.0003	0.00009	34.19

improves the learning rate of TRTRL to a level that surpasses the performance of RTRL, and is slightly lower than RTRL-SMD in accuracy. It further appears that, as the iteration count increases, the performance advantage of TRTRL-SMD also increases. This serves as a basic indication that, despite the partial sensitivities inherent to TRTRL, the gradient-based information propagated through the weights and activations of the network is sufficient to provide meaningful modeling capabilities.

Table I summarizes the mean square error (MSE), standard deviation (STD), and central processing unit (CPU) time for the four algorithms at the end of the training process. It is noted that while standard deviation values for all four algorithms are relatively low, for a comparable CPU time, TRTRL achieves the highest performance while TRTRL-SMD outperforms RTRL.

### B. Chaotic Time-Series Prediction

The task chosen was chaotic time-series prediction, whereby the networks are required to predict future values of the Mackey–Glass (MG) [20], [21] chaotic series, which has been extensively used as a benchmark. The MG series is based on the time-delayed differential equations

$$\frac{\partial x(t)}{\partial t} = -0.1x(t) + \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)}. \quad (31)$$

To obtain values at integer time points, the fourth-order Runge–Kutta method was used to find the numerical solution to the above MG equation. Here, we assume that the time step is 1,  $x(0) = 0.1$ ,  $\tau = 30$ , and  $x(t) = 0$  for  $t = 0$ .

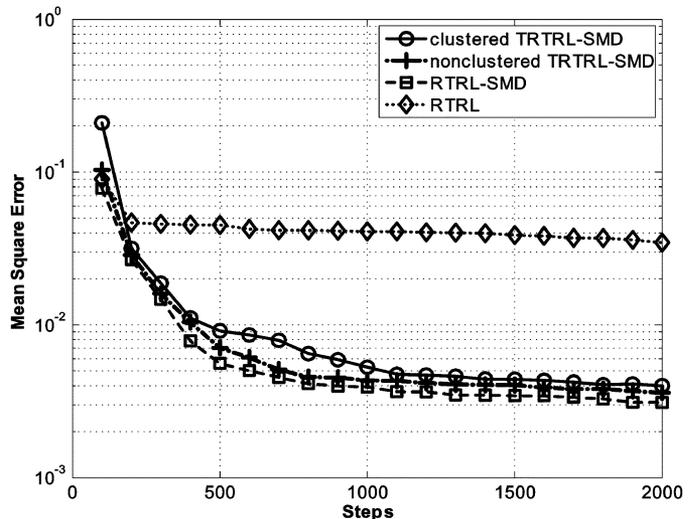


Fig. 3. Learning curves for the chaotic time-series prediction task, applied to a 16-neuron network running TRTRL-SMD, RTRL-SMD, and RTRL and a four-cluster network with four neurons each cluster running TRTRL-SMD.

The task is to predict the value of  $x(t+30)$  given the current input and internal state representation. The chaotic time-series prediction task was chosen because it is significantly more difficult for the networks to solve than the frequency doubler. The network topology was identical to the one used for the frequency doubler task. For nonclustered TRTRL-SMD, RTRL-SMD, and RTRL, we use 16 fully recurrent neurons, one bias input neuron whose (constant) value is 1, and a single linear output neuron. The same set of random initial weights was used for each network during every training run. The same settings of parameters for the three algorithm are used in this test as in the frequency doubler testbench. The clustered TRTRL-SMD network consists of four clusters with four neurons within each cluster. The neurons are fully connected within each cluster, while only the border neurons are connected between clusters. The output neuron is shared in order to propagate the gradient information to all the clusters (Fig. 1).

Fig. 3 illustrates the average learning curves for the four algorithms over 500 runs with each sampling point an average of 200 steps. This simulation task proved difficult for regular RTRL as the error did not drop below  $10^{-2}$ . In contrast, clustered TRTRL-SMD, nonclustered TRTRL-SMD, and RTRL-SMD have a clearly higher convergence rate, and higher degree of accuracy. It is the tendency of SMD to effectively adapt the step size in narrow error hypersurfaces that is attributed the substantial improvement in performance on this test case. Moreover, RTRL-SMD performs best because of its full connectivity. Further, as one would expect, the clustered TRTRL learns slower initially, due to the reduced connectivity, but it eventually reaches the same accuracy as with nonclustered TRTRL. Note that more neurons are needed for clustered TRTRL to achieve the same performance as nonclustered TRTRL, however, the extra neurons are well justified because the algorithm is more localized and the computational load is reduced.

Table II summarizes the MSE, STD, and CPU time for the four algorithms at the end of the training process in this test case. We observe that clustered TRTRL-SMD efficiently trades off accuracy for convergence rate, as it consumes the least CPU time during the learning process.

### C. Adaptation of Global Meta-Learning Rate

We next performed a comparison between TRTRL-SMD with constant and adaptive global meta-learning rate  $\mu$  on the chaotic series prediction task. For constant meta-learning rate, we chose  $\mu_{ij} = 0.5$  to ensure both performance and stability while for adaptive rate,  $\mu_{ij}$  was

TABLE II  
MSE, STD, AND CONVERGENCE IN CPU TIME FOR RTRL, RTRL-SMD, TRTRL-SMD, AND CLUSTERED TRTRL-SMD  
LEARNING ALGORITHMS IN THE CHAOTIC SERIES PREDICTION TEST

	MSE at Convergence	STD at Convergence	CPU time (sec)
RTRL	0.0346	0.00071	4.86
RTRL-SMD	0.0031	0.00056	10.43
TRTRL-SMD	0.0036	0.00037	1.96
clustered TRTRL-SMD	0.0040	0.00042	1.72

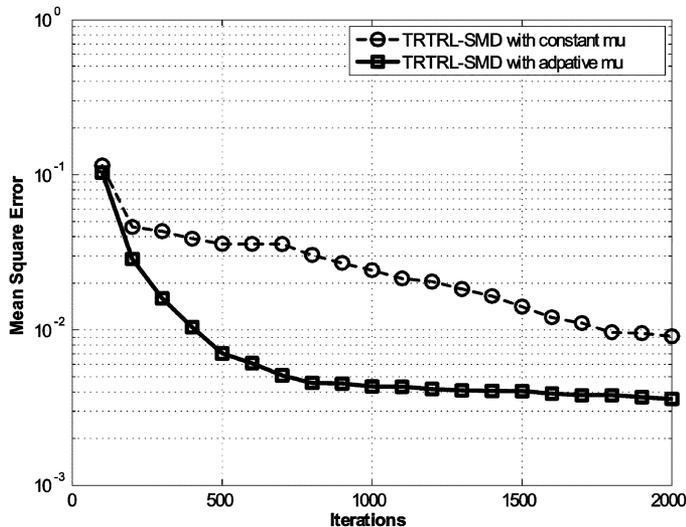


Fig. 4. Learning curves for the chaotic time-series prediction task, applied to a 16-neuron network running TRTRL-SMD with constant global meta-learning rate and adaptive global meta-learning rate.

varied within the bound  $[\mu_{\min} = 0.01, \mu_{\max} = 5]$ . The network structures and configuration of all other parameters are the same as in the previous section for both algorithms. Fig. 4 demonstrates the average learning curves for both algorithms over 500 runs. It clearly shows that TRTRL-SMD with adaptive learning rate achieves better performance in both convergence rate and accuracy than TRTRL-SMD with a constant learning rate.

## VII. CONCLUSION

In this brief, we presented a framework for substantially reducing the resource requirements of learning in recurrent neural network, while retaining high performance. The method is based on limiting the sensitivities of neuron activations to weights associated with either incoming or outgoing links, coupled with employing SMD—an efficient stochastic gradient-descent method. We also introduced clustered TRTRL-SMD, which further improves scalability and efficiency with minor compromise in performance. Based on standard testbench cases, it has been demonstrated through simulations that the performance of TRTRL-SMD exceeds that of RTRL, while speed and storage requirements are significantly reduced. On a broader scale, the methodology proposed in this brief can be applied to a wide range of other neural network architectures.

## ACKNOWLEDGMENT

The authors would like to thank D. Budik for the insightful discussions on TRTRL.

## REFERENCES

- [1] D. Nguyen and B. Widrow, "Neural networks for self-learning control system," *IEEE Control Syst. Mag.*, vol. 10, no. 3, pp. 18–23, Apr. 1990.
- [2] L. Jain, *Recurrent Neural Networks*. Boca Raton, FL: CRC Press, 2000.
- [3] D. V. Prokhorov, "Training recurrent neurocontrollers for real-time applications," *IEEE Trans. Neural Netw.*, vol. 18, no. 4, pp. 1003–1015, Jul. 2007.
- [4] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput.*, no. 1, pp. 270–280, 1989.
- [5] D. Zipser, "A subgrouping strategy that reduces complexity and speeds up learning in recurrent networks," *Neural Comput.*, no. 1, pp. 552–558, 1989.
- [6] N. Euliano and J. Principe, "Dynamic subgrouping in RTRL provides a faster  $O(n^2)$  algorithm," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, Istanbul, 2000, vol. 6, pp. 3418–3421.
- [7] O. D. Jesus and M. T. Hagan, "Backpropagation algorithms for a broad class of dynamic networks," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 14–27, Jan. 2007.
- [8] J. Schmidhuber, "A fixed size storage  $O(n^3)$  time complexity learning algorithm for fully recurrent continually running networks," *Neural Comput.*, vol. 4, pp. 243–248, 1992.
- [9] 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 *Proc. Neural Inf. Process. Syst.*, 1991, pp. 333–340.
- [10] V. S. Asirvadam, S. F. McLoone, and G. W. Irwin, "Memory efficient BFGS neural-network learning algorithms using MLP-network: A survey," in *Proc. IEEE Int. Conf. Control Appl.*, Sep. 2004, vol. 1, pp. 586–591.
- [11] D. Budik and I. Elhanany, "TRTRL: a localized resource-efficient learning algorithm for recurrent neural networks," in *Proc. IEEE Midwest Symp. Circuits Syst.*, Puerto Rico, Aug. 2006, pp. 371–374.
- [12] Z. Liu and I. Elhanany, "A scalable model-free recurrent neural network framework for solving POMDPs," in *Proc. IEEE Int. Symp. Approx. Dyn. Programm. Reinforcement Learn.*, Apr. 2007, pp. 119–126.
- [13] S. E. Fahlman, "An empirical study of learning speed in back-propagation networks," Computer Sci. Tech. Rep., 1988.
- [14] N. N. Schraudolph, "Local gain adaptation in stochastic gradient descent," in *Proc. 9th Int. Conf. Neural Netw.*, Sep. 1999, pp. 569–574.
- [15] N. N. Schraudolph, J. Yu, and D. Aberdeen, "Fast online policy gradient learning with SMD gain vector adaptation," in *Proc. 19th Annu. Conf. Neural Inf. Process. Syst.*, Vancouver, BC, Canada, Dec. 2005, pp. 1185–1192.
- [16] J. Kivinen and M. Warmuth, "Exponentiated gradient versus gradient descent for linear predictors," *Inf. Comput.*, no. 1, pp. 1–64, 1997.
- [17] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural Comput.*, vol. 6, no. 1, pp. 147–160, 1994.
- [18] G. D. Magoulas, M. N. Vrahatis, and G. S. Androulakis, "Improving the convergence of the backpropagation algorithm using learning rate adaptation methods," *Neural Comput.*, vol. 11, no. 7, pp. 1769–1796, 1999.
- [19] T. Tollenare, "Supersab: fast adaptive backpropagation with good scaling properties," *Neural Netw.*, vol. 3, no. 5, pp. 561–573, 1990.
- [20] M. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, pp. 287–289, Jul. 1977.
- [21] R. Croder, III, "Predicting the Mackey-Glass time-series with cascade-correlation learning," in *Connectionist Models Summer School Proceedings*, D. Touretzky, G. Hinton, and T. Sejnowski, Eds. Pittsburgh, PA: Carnegie Mellon Univ., 1990, pp. 117–123.