Communicated by Fernando Pineda

A Fixed Size Storage $O(n^3)$ Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks

Jürgen Schmidhuber
Department of Computer Science, University of Colorado, Campus Box 430, Boulder, CO 80309 USA

The real-time recurrent learning (RTRL) algorithm (Robinson and Fallside 1987; Williams and Zipser 1989) requires $O(n^4)$ computations per time step, where n is the number of noninput units. I describe a method suited for on-line learning that computes exactly the same gradient and requires fixed-size storage of the same order but has an average time complexity per time step of $O(n^3)$.

1 Introduction

There are two basic methods for performing steepest descent in fully recurrent networks with n noninput units and $m = O(n)$ input units. Backpropagation through time (BPTT) [e.g., Williams and Peng (1990)] requires potentially unlimited storage in proportion to the length of the longest training sequence but needs only $O(n^2)$ computations per time step. BPTT is the method of choice if training sequences are known to have less than $O(n)$ time steps. For training sequences involving many more time steps than n, for training sequences of unknown length, and for on-line learning in general one would like to have an algorithm with upper bounds for storage and for computations required per time step. Such an algorithm is the RTRL algorithm (Robinson and Fallside 1987; Williams and Zipser 1989). It requires only fixed-size storage of the order $O(n^3)$ but is computationally expensive: It requires $O(n^4)$ operations per time step.[1] The algorithm described herein[2] requires $O(n^3)$ storage, too. Every $O(n)$ time steps it requires $O(n^4)$ operations, but on all other time steps it requires only $O(n^2)$ operations. This cuts the average time complexity per time step to $O(n^3)$.

[1]Pineda has described another recurrent net algorithm that, as he states, "has some of the worst features of both algorithms" (Pineda 1990). His algorithm requires $\geq O(n^4)$ memory and $\geq O(n^4)$ computations per time step, if the number of time steps exceeds n.

[2]Since the acceptance of this paper for publication it has come to my attention that the same algorithm was derived by Ron Williams (Williams 1989; Williams and Zipser 1992).

2 The Algorithm

The notation will be similar to the notation of Williams and Peng (1990). U is the set of indices k such that at the discrete time step t the quantity $x_k(t)$ is the output of a noninput unit k in the network. I is the set of indices k such that $x_k(t)$ is an external input for input unit k at time t. $T(t)$ denotes the set of indices $k \in U$ for which there exists a specified target value $d_k(t)$ at time t. Each input unit has a directed connection to each noninput unit. Each noninput unit has a directed connection to each noninput unit. The weight of the connection from unit j to unit i is denoted by w_{ij}. To distinguish between different "instances" of w_{ij} at different times, we let $w_{ij}(t)$ denote a variable for the weight of the connection from unit j to unit i *at time* t. This is just for notational convenience: $w_{ij}(t) = w_{ij}$ for all t to be considered. One way to visualize the $w_{ij}(t)$ is to consider them as weights of connections to the tth noninput layer of a feedforward network constructed by "unfolding" the recurrent network in time [e.g., Williams and Peng (1990)]. A training sequence with $s+1$ time steps starts at time step 0 and ends at time step s. The algorithm below is of interest if $s \gg n$ (otherwise it is preferable to use BPTT).

For $k \in U$ we define

$$\begin{aligned} \mathrm{net}_k(0) = 0, \quad & \forall t \geq 0 : x_k(t) = f_k[\mathrm{net}_k(t)], \\ & \forall t > 0 : \quad \mathrm{net}_k(t+1) = \sum_{l \in U \cup I} w_{kl}(t+1) x_l(t) \end{aligned} \tag{2.1}$$

where f_k is a differentiable (usually semilinear) function. For all w_{ij} and for all $l \in U, t \geq 0$ we define

$$q^l_{ij}(t) = \frac{\partial \mathrm{net}_l(t)}{\partial w_{ij}} = \sum_{\tau=1}^{t} \frac{\partial \mathrm{net}_l(t)}{\partial w_{ij}(\tau)}$$

Furthermore we define

$$\begin{aligned} e_k(t) &= d_k(t) - x_k(t) \qquad \text{if } k \in T(t) \text{ and } 0 \text{ otherwise} \\ E(t) &= \frac{1}{2} \sum_{k \in U} [e_k(t)]^2, \qquad E^{\mathrm{total}}(t', t) = \sum_{\tau = t'+1}^{t} E(\tau) \end{aligned}$$

The algorithm is a cross between the BPTT and the RTRL algorithm. The description of the algorithm will be interleaved with its derivation and some comments concerning complexity. The basic idea is: Decompose the calculation of the gradient into blocks, each covering $O(n)$ time steps. For each block perform $n+1$ BPTT-like passes, one pass for calculating error derivatives, and n passes for calculating derivatives of the net-inputs to the n noninput units at the end of each block. Perform $n+1$ RTRL-like calculations for integrating the results of these BPTT-like passes into the results obtained from previous blocks.

The algorithm starts by setting the variable $t_0 \leftarrow 0$. t_0 represents the beginning of the current block. Note that for all possible l, w_{ij} : $q^l_{ij}(0) = 0$, $\partial E^{\text{total}}(0,0)/\partial w_{ij} = 0$. The main loop of the algorithm consists of five steps.

Step 1: Set $h \leftarrow O(n)$ (I recommend: $h \leftarrow n$).

The quantity $\partial E^{\text{total}}(0, t_0)/\partial w_{ij}$ for all w_{ij} is already known and $q^l_{ij}(t_0)$ is known for all appropriate l, i, j. There is an efficient way of computing the contribution of $E^{\text{total}}(0, t_0 + h)$ to the change in $w_{ij}, \Delta w_{ij}(t_0 + h)$:

$$\Delta w_{ij}(t_0 + h) = -\alpha \frac{\partial E^{\text{total}}(0, t_0 + h)}{\partial w_{ij}} = -\alpha \sum_{\tau=1}^{t_0+h} \frac{\partial E^{\text{total}}(0, t_0 + h)}{\partial w_{ij}(\tau)}$$

where α is a learning rate constant.

Step 2: Let the network run from time step t_0 to time step $t_0 + h$ according to the activation dynamics specified in equation 2.1. If it turns out that the current training sequence has less than $t_0 + h$ time steps (i.e., $h > s - t_0$) then $h \leftarrow s - t_0$. If $h = 0$ then EXIT.

Step 3: Perform a combination of a BPTT-like phase with an RTRL-like calculation for computing error derivatives as described next. We write

$$\begin{aligned}
\frac{\partial E^{\text{total}}(0, t_0 + h)}{\partial w_{ij}} &= \frac{\partial E^{\text{total}}(0, t_0)}{\partial w_{ij}} + \frac{\partial E^{\text{total}}(t_0, t_0 + h)}{\partial w_{ij}} \\
&= \frac{\partial E^{\text{total}}(0, t_0)}{\partial w_{ij}} + \sum_{\tau=1}^{t_0+h} \frac{\partial E^{\text{total}}(t_0, t_0 + h)}{\partial w_{ij}(\tau)} \\
&= \frac{\partial E^{\text{total}}(0, t_0)}{\partial w_{ij}} + \sum_{\tau=1}^{t_0} \frac{\partial E^{\text{total}}(t_0, t_0 + h)}{\partial w_{ij}(\tau)} \\
&\quad + \sum_{\tau=t_0+1}^{t_0+h} \frac{\partial E^{\text{total}}(t_0, t_0 + h)}{\partial w_{ij}(\tau)} \qquad (2.2)
\end{aligned}$$

The first term of equation 2.2 is already known. Consider the third term:

$$\sum_{\tau=t_0+1}^{t_0+h} \frac{\partial E^{\text{total}}(t_0, t_0 + h)}{\partial w_{ij}(\tau)} = - \sum_{\tau=t_0+1}^{t_0+h} \delta_i(\tau) x_j(\tau - 1)$$

where

$$\delta_i(\tau) = -\frac{\partial E^{\text{total}}(t_0, t_0 + h)}{\partial \text{net}_i(\tau)}$$

For a given t_0, $\delta_i(\tau)$ can be computed for all $i \in U, t_0 \leq \tau \leq t_0 + h$ with a single h step BPTT-pass of the order $O(hn^2)$ operations:

$$\begin{aligned}
\delta_i(\tau) &= f'_i[\text{net}_i(\tau)] e_i(\tau) \qquad \text{if } \tau = t_0 + h \\
\delta_i(\tau) &= f'_i[\text{net}_i(\tau)] \left[e_i(\tau) + \sum_{l \in U} w_{li} \delta_l(\tau + 1) \right] \qquad \text{if } t_0 \leq \tau < t_0 + h
\end{aligned}$$

What remains is the computation of the second term of equation 2.2 for all w_{ij}, which requires $O(n^3)$ operations:

$$\begin{aligned}\sum_{\tau=1}^{t_0} \frac{\partial E^{\text{total}}(t_0, t_0+h)}{\partial w_{ij}(\tau)} &= \sum_{\tau=1}^{t_0} \sum_{k\in U} \frac{\partial E^{\text{total}}(t_0, t_0+h)}{\partial \text{net}_k(t_0)} \frac{\partial \text{net}_k(t_0)}{\partial w_{ij}(\tau)} \\ &= \sum_{k\in U} -\delta_k(t_0) \sum_{\tau=1}^{t_0} \frac{\partial \text{net}_k(t_0)}{\partial w_{ij}(\tau)} = -\sum_{k\in U} \delta_k(t_0) q_{ij}^k(t_0)\end{aligned}$$

Step 4: To compute $q_{ij}^l(t_0+h)$ for all possible l, i, j, perform n combinations of a BPTT-like phase with an RTRL-like calculation (one such combination for each l) for computing as follows:

$$\begin{aligned} q_{ij}^l(t_0+h) &= \frac{\partial \text{net}_l(t_0+h)}{\partial w_{ij}} = \sum_{\tau=1}^{t_0+h} \frac{\partial \text{net}_l(t_0+h)}{\partial w_{ij}(\tau)} \\ &= \sum_{\tau=1}^{t_0} \frac{\partial \text{net}_l(t_0+h)}{\partial w_{ij}(\tau)} \\ &+ \sum_{\tau=t_0+1}^{t_0+h} \frac{\partial \text{net}_l(t_0+h)}{\partial w_{ij}(\tau)} \\ &= \sum_{\tau=1}^{t_0} \sum_{k\in U} \frac{\partial \text{net}_l(t_0+h)}{\partial \text{net}_k(t_0)} \frac{\partial \text{net}_k(t_0)}{\partial w_{ij}(\tau)} \\ &+ \sum_{\tau=t_0+1}^{t_0+h} \frac{\partial \text{net}_l(t_0+h)}{\partial \text{net}_i(\tau)} \frac{\partial \text{net}_i(\tau)}{\partial w_{ij}(\tau)} \\ &= \sum_{k\in U} \gamma_{lk}(t_0) h_{ij}^k(t_0) + \sum_{\tau=t_0+1}^{t_0+h} \gamma_{li}(\tau) x_j(\tau-1) \end{aligned} \tag{2.3}$$

where

$$\gamma_{lk}(\tau) = \frac{\partial \text{net}_l(t_0+h)}{\partial \text{net}_k(\tau)}$$

For a given t_0, a given $l \in U$ and for all $i \in U, t_0 \le \tau \le t_0+h$ the quantity $\gamma_{li}(\tau)$ can be computed with a single h step BPTT-like operation of the order $O(hn^2)$:

$$\text{if } \tau = t_0+h: \qquad \text{if } l = i \text{ then } \gamma_{li}(\tau) = 1 \text{ else } \gamma_{li}(\tau) = 0$$

$$\text{if } t_0 \le \tau < t_0+h: \qquad \gamma_{li}(\tau) = f_i'[\text{net}_i(\tau)] \sum_{a\in U} w_{ai}\gamma_{la}(\tau+1)$$

For a given l, the computation of equation 2.3 for all w_{ij} requires $O(n^3 + hn^2)$ operations. Therefore Step 3 and Step 4 together require $(n+1)O(hn^2+n^3)$ operations *spread over h time steps*. Since $h = O(n)$, $O(n^4)$ computations are spread over $O(n)$ time steps. *This implies an average of $O(n^3)$ computations per time step.*

The final step of the algorithm's main loop is

Step 5: Set $t_0 \leftarrow t_0 + h$ and go to Step 1.

The off-line version of the algorithm waits until the end of an episode (which needs not be known in advance) before performing weight changes. An on-line version performs weight changes each time Step 4 is completed.

As formulated above, the algorithm needs $O(n^4)$ computations at its peak, every nth time step. Nothing prevents us, however, from distributing these $O(n^4)$ computations more evenly over n time steps. One way of achieving this is to perform one of the n BPTT-like phases of Step 4 at each time step of the next "block" of n time steps.

3 Concluding Remarks

Like the RTRL-algorithm the method needs a fixed amount of storage of the order $O(n^3)$. Like the RTRL-algorithm [but unlike the methods described in Williams and Peng (1990) and Zipser (1989)] the algorithm computes the exact gradient. Since it is $O(n)$ times faster than RTRL, it should be preferred.

Following the argumentation in Williams and Peng (1990), continuous time versions of BPTT and RTRL (Pearlmutter 1989; Gherrity 1989) can serve as a basis for a correspondingly efficient continuous time version of the algorithm presented here (by means of Euler discretization).

Many typical environments produce input sequences that have both local and more global temporal structure. For instance, input sequences are often hierarchically organized (e.g., speech). In such cases, sequence-composing algorithms (Schmidhuber 1991a,b) can provide superior alternatives to pure gradient-based algorithms.

Acknowledgments

Thanks to Mike Mozer, Bernd Schürmann, and Daniel Prelinger for providing useful comments on an earlier draft of this paper.

This research was supported by NSF PYI award IRI-9058450, grant 90-21 from the James S. McDonnell Foundation, and DEC external research grant 1250 to Michael C. Mozer.

References

Gherrity, M. 1989. A learning algorithm for analog fully recurrent neural networks. *IEEE/INNS Int. Joint Conf. Neural Networks, San Diego* **1**, 643–644.

Pearlmutter, B. A. 1989. Learning state space trajectories in recurrent neural networks. *Neural Comp.* **1**, 263–269.

Pineda, F. J. 1990. Time dependent adaptive neural networks. In *Advances in Neural Information Processing Systems* 2, D. S. Touretzky, ed., pp. 710–718. Morgan Kaufmann, San Mateo, CA.

Robinson, A. J., and Fallside, F. 1987. *The utility driven dynamic error propagation network.* Tech. Rep. CUED/F-INFENG/TR.1, Cambridge University Engineering Department.

Schmidhuber, J. H. 1991a. Adaptive decomposition of time. In *Artificial Neural Networks,* T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, eds., pp. 909–914. Elsevier Science Publishers B.V., North-Holland.

Schmidhuber, J. H. 1991b. Learning complex, extended sequences using the principle of history compression. *Neural Comp.* **4**, 234–242.

Williams, R J. 1989. Complexity of exact gradient computation algorithms for recurrent neural networks. Tech. Rep. NU-CCS-89-27, Boston: Northeastern University, College of Computer Science.

Williams, R. J., and Peng, J. 1990. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Comp.* **4**, 491–501.

Williams, R. J., and Zipser, D. 1989. Experimental analysis of the real-time recurrent learning algorithm. *Connection Sci.* **1**(1), 87–111.

Williams, R. J., and Zipser, D. 1992. Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Backpropagation: Theory, Architectures and Applications,* Y. Chauvin and D. E. Rumelhart, eds. Hillsdale, NJ: Erlbaum.

Zipser, D. 1989. A subgrouping strategy that reduces learning complexity and speeds up learning in recurrent networks. *Neural Comp.* **1**, 552–558.

Received 16 May 1991; accepted 20 September 1991.