*LETTER* Communicated by Kenji Doya

# Learning to Forget: Continual Prediction with LSTM

**Felix A. Gers**
**Jürgen Schmidhuber**
**Fred Cummins**
*IDSIA, 6900 Lugano, Switzerland*

**Long short-term memory (LSTM; Hochreiter & Schmidhuber, 1997) can solve numerous tasks not solvable by previous learning algorithms for recurrent neural networks (RNNs). We identify a weakness of LSTM networks processing continual input streams that are not a priori segmented into subsequences with explicitly marked ends at which the network's internal state could be reset. Without resets, the state may grow indefinitely and eventually cause the network to break down. Our remedy is a novel, adaptive "forget gate" that enables an LSTM cell to learn to reset itself at appropriate times, thus releasing internal resources. We review illustrative benchmark problems on which standard LSTM outperforms other RNN algorithms. All algorithms (including LSTM) fail to solve continual versions of these problems. LSTM with forget gates, however, easily solves them, and in an elegant way.**

## 1 Introduction

Recurrent neural networks (RNNs) constitute a very powerful class of computational models, capable of instantiating almost arbitrary dynamics. The extent to which this potential can be exploited is, however, limited by the effectiveness of the training procedure applied. Gradient-based methods–back-propagation through time (Williams & Zipser, 1992; Werbos, 1988) or real-time recurrent learning (Robinson & Fallside, 1987; Williams & Zipser, 1992) and their combination (Schmidhuber, 1992)—share an important limitation. The temporal evolution of the path integral over all error signals "flowing back in time" exponentially depends on the magnitude of the weights (Hochreiter, 1991). This implies that the backpropagated error (Pearlmutter, 1995) quickly either vanishes or blows up (Hochreiter & Schmidhuber, 1997; Bengio, Simard, & Frasconi, 1994). Hence standard RNNs fail to learn in the presence of time lags greater than 5–10 discrete time steps between relevant input events and target signals. The vanishing error problem casts doubt on whether standard RNNs can indeed exhibit significant practical advantages over time-window-based feedforward networks.

*Neural Computation* **12, 2451–2471** (2000) 

A recent model, long short-term memory (LSTM) (Hochreiter & Schmidhuber, 1997), is not affected by this problem. LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing constant error flow through constant error carousels (CECs) within special units called cells. Multiplicative gate units learn to open and close access to the cells. LSTM's learning algorithm is local in space and time; its computational complexity per time step and weight is $O(1)$. It solves complex long-time-lag tasks that have never been solved by previous RNN algorithms. (See Hochreiter & Schmidhuber, 1997, for a comparison of LSTM to alternative approaches.)

In this article, however, we show that even LSTM fails to learn to process certain very long or continual time series correctly that are not a priori segmented into appropriate training subsequences with clearly defined beginnings and ends. The problem is that a continual input stream eventually may cause the internal values of the cells to grow without bound, even if the repetitive nature of the problem suggests they should be reset occasionally. This article presents a remedy.

Although we present a specific solution to the problem of forgetting in LSTM networks, we recognize that any training procedure for RNNs that is powerful enough to span long time lags must also address the issue of forgetting in short-term memory (unit activations). We know of no other current training method for RNNs that is sufficiently powerful to have encountered this problem.

Section 2 briefly summarizes LSTM and explains its weakness in processing continual input streams. Section 3 introduces a remedy called forget gates. Forget gates learn to reset memory cell contents once they are not needed anymore. Forgetting may occur rhythmically or in an input-dependent fashion. Section 4 derives a gradient-based learning algorithm for the LSTM extension with forget gates. Section 5 describes experiments: we transform well-known benchmark problems into more complex, continual tasks, report the performance of various RNN algorithms, and analyze and compare the networks found by standard LSTM and extended LSTM.

## 2 Standard LSTM

The basic unit in the hidden layer of an LSTM network is the memory block, which contains one or more memory cells and a pair of adaptive, multiplicative gating units that gate input and output to all cells in the block. Each memory cell has at its core a recurrently self-connected linear unit called the constant error carousel (CEC), whose activation we call the cell state. The CECs solve the vanishing error problem: in the absence of new input or error signals to the cell, the CEC's local error backflow remains constant, neither growing nor decaying. The CEC is protected from both forward-flowing activation and backward-flowing error by the input and

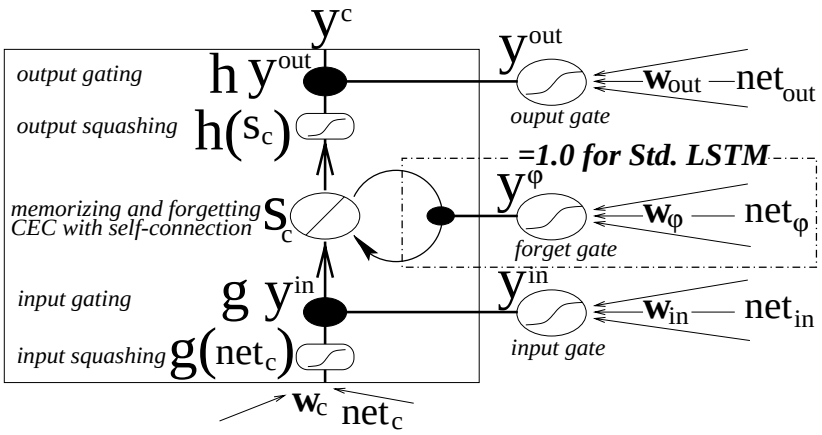


Figure 1: The standard LSTM cell has a linear unit with a recurrent self-connection with weight 1.0 (CEC). Input and output gates regulate read and write access to the cell whose state is denoted $s_c$. The function $g$ squashes the cell's input; $h$ squashes the cell's output. See the text for details.

output gates, respectively. When gates are closed (activation around zero), irrelevant inputs and noise do not enter the cell, and the cell state does not perturb the remainder of the network. Figure 1 shows a memory block with a single cell. The cell state, $s_c$, is updated based on its current state and three sources of input: $net_c$ is input to the cell itself, while $net_{in}$ and $net_{out}$ are inputs to the input and output gates.

We consider discrete time steps $t = 1, 2, \ldots$. A single step involves the update of all units (forward pass) and the computation of error signals for all weights (backward pass). Input gate activation $y^{in}$ and output gate activation $y^{out}$ are computed as follows:

$$net_{out_j}(t) = \sum_m w_{out_j m}\, y^m(t-1),\ y^{out_j}(t) = f_{out_j}(net_{out_j}(t)), \tag{2.1}$$

$$net_{in_j}(t) = \sum_m w_{in_j m}\, y^m(t-1),\ y^{in_j}(t) = f_{in_j}(net_{in_j}(t)). \tag{2.2}$$

Throughout this article, $j$ indexes memory blocks; $v$ indexes memory cells in block $j$, such that $c_j^v$ denotes the $v$th cell of the $j$th memory block; $w_{lm}$ is the weight on the connection from unit $m$ to unit $l$. Index $m$ ranges over all source units, as specified by the network topology. For gates, $f$ is a logistic sigmoid with range $[0, 1]$. Net input to the cell itself is squashed by $g$, a centered logistic sigmoid function with range $[-2, 2]$.

The internal state of memory cell $s_c(t)$ is calculated by adding the squashed, gated input to the state at the previous time step $s_c(t-1)$ ($t > 0$):

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m}\, y^m(t-1), \quad s_{c_j^v}(t) = s_{c_j^v}(t-1) + y^{in_j}(t)\, g(net_{c_j^v}(t)), \quad (2.3)$$

with $s_{c_j^v}(0) = 0$. The cell output $y^c$ is calculated by squashing the internal state $s_c$ via the output squashing function $h$ and then multiplying (gating) it by the output gate activation $y^{out}$:

$$y^{c_j^v}(t) = y^{out_j}(t)\, h(s_{c_j^v}(t)). \quad (2.4)$$

$h$ is a centered sigmoid with range $[-1, 1]$.

Finally, assuming a layered network topology with a standard input layer, a hidden layer consisting of memory blocks, and a standard output layer, the equations for the output units $k$ are:

$$net_k(t) = \sum_m w_{km}\, y^m(t-1), \; y^k(t) = f_k(net_k(t)), \quad (2.5)$$

where $m$ ranges over all units feeding the output units (typically all cells in the hidden layer, the input units, but not the memory block gates). As squashing function $f_k$, we again use the logistic sigmoid, range $[0, 1]$. All equations except for equation 2.3 will remain valid for extended LSTM with forget gates.

Hochreiter and Schmidhuber (1997) provide details of standard LSTM's backward pass. Essentially, as in truncated backpropagation through time (BPTT), errors arriving at net inputs of memory blocks and their gates do not get propagated back further in time, although they do serve to change the incoming weights. An error signal arriving at a memory cell output is scaled by the output gate and the output nonlinearity $h$; it then enters the memory cell's linear CEC, where it can flow back indefinitely without ever being changed (this is why LSTM can bridge arbitrary time lags between input events and target signals). Only when the error escapes from the memory cell through an opening input gate and the additional input nonlinearity $g$ does it get scaled once more, and then serves to change incoming weights before being truncated. (Details of extended LSTM's backward pass are discussed in section 3.2.)

**2.1 Limits of Standard LSTM.** LSTM allows information to be stored across arbitrary time lags and error signals to be carried far back in time. This potential strength, however, can contribute to a weakness in some situations. The cell states $s_c$ often tend to grow linearly during the presentation of a time series (the nonlinear aspects of sequence processing are left to the squashing functions and the highly nonlinear gates). If we present a continuous input

stream, the cell states may grow in unbounded fashion, causing saturation of the output squashing function, $h$. This happens even if the nature of the problem suggests that the cell states should be reset occasionally, such as at the beginnings of new input sequences (whose starts, however, are not explicitly indicated by a teacher). Saturation will make $h$'s derivative vanish, thus blocking incoming errors, and make the cell output equal the output gate activation; that is, the entire memory cell will degenerate into an ordinary BPTT unit, so that the cell will cease functioning as a memory. The problem did not arise in the experiments reported by Hochreiter and Schmidhuber (1997) because cell states were explicitly reset to zero before the start of each new sequence.

How can we solve this problem without losing LSTM's advantages over time-delay neural networks (TDNN) (Waibel, 1989) or NARX (nonlinear autoregressive models with exogenous inputs) (Lin, Horne, Tiño, & Giles, 1996), which depend on a priori knowledge of typical time lag sizes? The standard technique of weight decay, which helps to contain the level of overall activity within the network, was found to generate solutions that were particularly prone to unbounded state growth. Variants of focused backpropagation (Mozer, 1989) also do not work well. These let the internal state decay via a self-connection whose weight is smaller than 1. But there is no principled way of designing appropriate decay constants. A potential gain for some tasks is paid for by a loss of ability to deal with arbitrary, unknown causal delays between inputs and targets. In fact, state decay does not significantly improve experimental performance (see "state decay" in Table 2).

Of course, we might try to "teacher force" (Jordan, 1986; Doya & Yoshizawa, 1989) the internal states $s_c$ by resetting them once a new training sequence starts. But this requires an external teacher who knows how to segment the input stream into training subsequences. We are precisely interested, however, in those situations where there is no a priori knowledge of this kind.

## 3 Solution: Forget Gates

Our solution to the problem above is to use adaptive forget gates, which learn to reset memory blocks once their contents are out of date and hence useless. By resets, we do not mean only immediate resets to zero but also gradual resets corresponding to slowly fading cell states. More specifically, we replace standard LSTM's constant CEC weight 1.0 by the multiplicative forget gate activation $y^{\varphi}$. (See Figure 1.)

**3.1 Forward Pass of Extended LSTM with Forget Gates.** The forget gate activation $y^{\varphi}$ is calculated like the activations of the other gates (see

equations 2.1 and 2.2):

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m}\, y^m(t-1); \quad y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)). \tag{3.1}$$

Here $net_{\varphi_j}$ is the input from the network to the forget gate. We use the logistic sigmoid with range $[0, 1]$ as squashing function $f_{\varphi_j}$. Its output becomes the weight of the self-recurrent connection of the internal state $s_c$ in equation 2.3. The revised update equation for $s_c$ in the extended LSTM algorithm is (for $t > 0$):

$$s_{c_j^v}(t) = y^{\varphi_j}(t)\, s_{c_j^v}(t-1) + y^{in_j}(t)\, g(net_{c_j^v}(t)), \tag{3.2}$$

with $s_{c_j^v}(0) = 0$. Extended LSTM's full-forward pass is obtained by adding equations 3.1 to those in section 2 and replacing equation 2.3 by 3.2.

Bias weights for LSTM gates are initialized with negative values for input and output gates (see Hochreiter & Schmidhuber, 1997, for details) and positive values for forget gates. This implies (compare equations 3.1 and 3.2) that in the beginning of the training phase, the forget gate activation will be almost 1.0, and the entire cell will behave like a standard LSTM cell. It will not explicitly forget anything until it has learned to forget.

**3.2 Backward Pass of Extended LSTM with Forget Gates.** LSTM's backward pass (see Hochreiter & Schmidhuber, 1997, for details) is an efficient fusion of slightly modified, truncated BPTT (e.g., Williams & Peng, 1990) and a customized version of real time recurrent learning (RTRL) (e.g., Robinson & Fallside, 1987). Output units use BPTT; output gates use slightly modified, truncated BPTT. Weights to cells, input gates, and the novel forget gates, however, use a truncated version of RTRL. Truncation means that all errors are cut off once they leak out of a memory cell or gate, although they do serve to change the incoming weights. The effect is that the CECs are the only part of the system through which errors can flow back forever. This makes LSTM's updates efficient without significantly affecting learning power: error flow outside cells tends to decay exponentially anyway (Hochreiter, 1991). In the equations that follow, $\stackrel{tr}{=}$ will indicate where we use error truncation; unless otherwise indicated, we assume for simplicity only a single cell per block.

We start with the usual squared error objective function based on targets $t^k$,

$$E(t) = \frac{1}{2} \sum_k e_k(t)^2; \quad e_k(t) := t^k(t) - y^k(t), \tag{3.3}$$

where $e_k$ denotes the externally injected error. We minimize $E$ via gradient descent by adding weight changes $\Delta w_{lm}$ to the weights $w_{lm}$ (from unit $m$ to

unit $l$) using learning rate $\alpha$:

$$\begin{aligned}\Delta w_{lm}(t) &= -\alpha \frac{\partial E(t)}{\partial w_{lm}} = -\alpha \frac{\partial E(t)}{\partial y^k(t)} \frac{\partial y^k(t)}{\partial w_{lm}} = \alpha \sum_k e_k(t) \frac{\partial y^k(t)}{\partial w_{lm}} \\ &\stackrel{tr}{=} \alpha \sum_k e_k(t) \frac{\partial y^k(t)}{\partial y^l(t)} \frac{\partial y^l(t)}{\partial net_l(t)} \underbrace{\frac{\partial net_l(t)}{\partial w_{lm}}}_{=y^m(t-1)} \\ &= \alpha \underbrace{\frac{\partial y^l(t)}{\partial net_l(t)} \left( \sum_k \frac{\partial y^k(t)}{\partial y^l(t)} e_k(t) \right)}_{=:\delta_l(t)} y^m(t-1). \end{aligned} \tag{3.4}$$

For an arbitrary output unit ($l = k$), the sum in equation 3.4 vanishes. By differentiating equation 2.5, we obtain the usual backpropagation weight changes for the output units:

$$\frac{\partial y^k(t)}{\partial net_k(t)} = f_k'(net_k(t)) \Longrightarrow \delta_k(t) = f_k'(net_k(t))\, e_k(t). \tag{3.5}$$

To compute the weight changes for the output gates $\Delta w_{out_j m}$, we set ($l = out$) in equation 3.4. The resulting terms can be determined by differentiating equations 2.1, 2.4, and 2.5:

$$\frac{\partial y^{out_j}(t)}{\partial net_{out_j}(t)} = f_{out_j}'(net_{out_j}(t)),\ \frac{\partial y^k(t)}{\partial y^{out_j}(t)} e_k(t) = h(s_{c_j^v}(t))\, w_{kc_j^v}\, \delta_k(t).$$

Inserting both terms in equation 3.4 gives $\delta_{out_j}^v$, the contribution of the block's $v$th cell to $\delta_{out_j}$. Because every cell in a memory block contributes to the weight change of the output gate, we have to sum over all cells $v$ in block $j$ to obtain the total $\delta_{out_j}$ of the $j$th memory block (with $S_j$ cells):

$$\delta_{out_j}(t) = f_{out_j}'(net_{out_j}(t)) \left( \sum_{v=1}^{S_j} h(s_{c_j^v}(t)) \sum_k w_{kc_j^v}\, \delta_k(t) \right). \tag{3.6}$$

Equations 3.4 through 3.6 define the weight changes for output units and output gates of memory blocks. Their derivation was almost standard BPTT, with error signals truncated once they leave memory blocks (including its gates). This truncation does not affect LSTM's long time lag capabilities but is crucial for all equations of the backward pass and should be kept in mind.

For weights to cell, input gate, and forget gate, we adopt an RTRL-oriented perspective by first stating the influence of a cell's internal state $s_{c_j^v}$ on the error and then analyzing how each weight to the cell or the block's gates contributes to $s_{c_j^v}$. So we split the gradient in a way different from the one used in equation 3.4:

$$\Delta w_{lm}(t) = -\alpha \frac{\partial E(t)}{\partial w_{lm}} \stackrel{tr}{=} -\alpha \underbrace{\frac{\partial E(t)}{\partial s_{c_j^v}(t)}}_{=:-e_{s_{c_j^v}}(t)} \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} = \alpha\, e_{s_{c_j^v}}(t) \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}}. \tag{3.7}$$

These terms are the internal state error $e_{s_{c_j^v}}$ and a partial $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ of $s_{c_j^v}$ with respect to weights $w_{lm}$ feeding the cell $c_j^v$ ($l = c_j^v$) or the block's input gate ($l = in$) or the block's forget gate ($l = \varphi$), as all these weights contribute to the calculation of $s_{c_j^v}(t)$. We treat the partial for the internal states error $e_{s_{c_j^v}}$ analogously to equation 3.4 and obtain:

$$e_{s_{c_j^v}}(t) := -\frac{\partial E(t)}{\partial s_{c_j^v}(t)} \stackrel{tr}{=} -\frac{\partial E(t)}{\partial y^k(t)} \frac{\partial y^k(t)}{\partial y^{c_j^v}(t)} \frac{\partial y^{c_j^v}(t)}{\partial s_{c_j^v}(t)} = \frac{\partial y^{c_j^v}}{\partial s_{c_j^v}(t)} \sum_k \underbrace{\frac{\partial y^k(t)}{\partial y^{c_j^v}(t)} e_k(t)}_{=w_{c_j^v l}\delta_k(t)}.$$

Differentiating the forward pass equation, 2.4, $\frac{\partial y^{c_j^v}}{\partial s_{c_j^v}(t)} = y^{out_j}(t)\, h'(s_{c_j^v}(t))$, we obtain:

$$e_{s_{c_j^v}}(t) = y^{out_j}(t)\, h'(s_{c_j^v}(t)) \left( \sum_k w_{k c_j^v}\, \delta_k(t) \right). \tag{3.8}$$

This internal state error needs to be calculated for each memory cell. To calculate the partial $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ in equation 3.7, we differentiate equation 3.2 and obtain a sum of four terms:

$$\begin{aligned} \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} = & \underbrace{\frac{\partial s_{c_j^v}(t-1)}{\partial w_{lm}} y^{\varphi_j}(t)}_{\neq 0 \text{ for all } l \in \{\varphi, in, c_j^v\}} + \underbrace{y^{in_j}(t) \frac{\partial g(net_{c_j^v}(t))}{\partial w_{lm}}}_{\neq 0 \text{ for } l = c_j^v \text{ (cell)}} \\ & + \underbrace{g(net_{c_j^v}(t)) \frac{\partial y^{in_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = in \text{ (input gate)}} + \underbrace{s_{c_j^v}(t-1) \frac{\partial y^{\varphi_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = \varphi \text{ (forget gate)}}. \end{aligned} \tag{3.9}$$

Differentiating the forward pass equations 2.3, 2.2, and 3.1 for $g$, $y^{in}$, and

$y^{\varphi}$, we can substitute the unresolved partials and split the expression on the right-hand side of equation 3.9 into three separate equations for the cell ($l = c_j^v$), the input gate ($l = in$), and the forget gate ($l = \varphi$):

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} y^{\varphi_j}(t) + g'(net_{c_j^v}(t))\, y^{in_j}(t)\, y^m(t-1), \tag{3.10}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} y^{\varphi_j}(t) + g(net_{c_j^v}(t))\, f'_{in_j}(net_{in_j}(t))\, y^m(t-1), \tag{3.11}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} y^{\varphi_j}(t) + s_{c_j^v}(t-1)\, f'_{\varphi_j}(net_{\varphi_j}(t))\, y^m(t-1). \tag{3.12}$$

Furthermore the initial state of network does not depend on the weights, so we have

$$\frac{\partial s_{c_j^v}(t=0)}{\partial w_{lm}} = 0 \quad \text{for } l \in \{\varphi, in, c_j^v\}. \tag{3.13}$$

Note that the recursions in equations 3.10 through 3.12 depend on the actual activation of the block's forget gate. When the activation goes to zero, not only the cell's state but also the partials are reset (forgetting includes forgiving previous mistakes). Every cell needs to keep a copy of each of these three partials and update them at every time step.

We can insert the partials in equation 3.7 and calculate the corresponding weight updates, with the internal state error $e_{s_{c_j^v}}(t)$ given by equation 3.8. The difference between updates of weights to a cell itself ($l = c_j^v$) and updates of weights to the gates is that changes to weights to the cell $\Delta w_{c_j^v m}$ depend on only the partials of this cell's own state:

$$\Delta w_{c_j^v m}(t) = \alpha e_{s_{c_j^v}}(t) \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}}. \tag{3.14}$$

To update the weights of the input gate and the forget gate, however, we have to sum over the contributions of all cells in the block:

$$\Delta w_{lm}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \text{ for } l \in \{\varphi, in\}. \tag{3.15}$$

The equations necessary to implement the backward pass are 3.4 through 3.6, 3.8, and 3.10 through 3.15.

The appendix summarizes the complete LSTM algorithm (forward and backward pass) in pseudocode.

**3.3 Complexity.** To calculate the computational complexity of extended LSTM, we take into account that weights to input gates and forget gates cause more expensive updates than others, because each such weight directly affects all the cells in its memory block. We evaluate a rather typical topology used in the experiments (see Figure 3). All memory blocks have the same size; gates have no outgoing connections; output units and gates have a bias connection (from a unit whose activation is always 1.0); other connections to output units stem from memory blocks only; the hidden layer is fully connected. Let $B$, $S$, $I$, $K$ denote the numbers of memory blocks, memory cells in each block, input units, and output units, respectively. We find the update complexity per time step to be:

$$W_c = \underbrace{(B \cdot (S + 2S + 1)) \cdot (B \cdot (S + 2)) + B \cdot (2S + 1)}_{\text{recurrent connections and bias}} + \underbrace{(B \cdot S + 1) \cdot K}_{\text{to output}} + \underbrace{(B \cdot (S + 2S + 1)) \cdot I}_{\text{from input}} . \tag{3.16}$$

Hence LSTM's update complexity per time step and weight is of order $O(1)$, essentially the same as for a fully connected BPTT recurrent network. Storage complexity per weight is also $O(1)$, as the last time step's partials from equations 3.10 through 3.12 are all that need to be stored for the backward pass. So the storage complexity does not depend on the length of the input sequence. Hence, extended LSTM is local in space and time, according to Schmidhuber's definition (1989), just like standard LSTM.

## 4 Experiments

**4.1 Continual Embedded Reber Grammar Problem.** To generate an infinite input stream, we extend the well-known embedded Reber grammar (ERG) benchmark problem, Smith & Zipser 1989; Cleeremans, Servan-Schreiber, & McClelland, 1989; Fahlman, 1991; Hochreiter & Schmidhuber, 1997). Consider Figure 2.

*4.1.1 ERG.* The traditional method starts at the left-most node of the ERG graph and sequentially generates finite symbol strings (beginning with the empty string) by following edges and appending the associated symbols

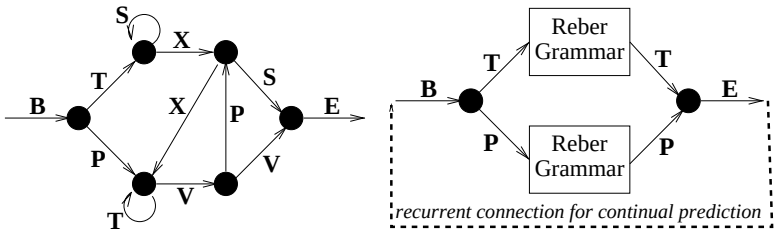


Figure 2: Transition diagrams for standard (left) and embedded (right) Reber grammars. The dashed line indicates the continual variant.

to the current string until the right-most node is reached. Edges are chosen randomly if there is a choice (probability $= 0.5$).

Input and target symbols are represented by seven-dimensional binary vectors, each component standing for one of the seven possible symbols. Hence the network has seven input units and seven output units. The task is to read strings, one symbol at a time, and to predict continually the next possible symbol(s). Input vectors have exactly one nonzero component. Target vectors may have two, because sometimes there is a choice of two possible symbols at the next step. A prediction is considered correct if the mean squared error at each of the seven output units is below 0.49 (error signals occur at every time step).

To predict correctly the symbol before the last (**T** or **P**) in an ERG string, the network has to remember the second symbol (also **T** or **P**) without confusing it with identical symbols encountered later. The minimal time lag is 7 (at the limit of what standard recurrent networks can manage); time lags have no upper bound, though. The expected length of a string generated by an ERG is 11.5 symbols. The length of the longest string in a set of $N$ nonidentical strings is proportional to $\log N$ (Gers, Schmidhuber, & Cummins, 1999). For the training and test sets used in our experiments, the expected value of the longest string is greater than 50.

Table 1 summarizes the performance of previous RNNs on the standard ERG problem (testing involved a test set of 256 ERG test strings). Only LSTM always learns to solve the task. Even when we ignore the unsuccessful trials of the other approaches, LSTM learns much faster.

*4.1.2 CERG.* Our more difficult continual variant of the ERG problem (CERG) does not provide information about the beginnings and ends of symbol strings. Without intermediate resets, the network is required to learn, in an on-line fashion, from input streams consisting of concatenated ERG strings. Input streams are stopped as soon as the network makes an incorrect prediction or the $10^5$th successive symbol has occurred. Learning and testing alternate: after each training stream, we freeze the weights and

Table 1: Standard Embedded Reber Grammar (ERG): Percentage of Successful Trials and Number of Sequence Presentations until Success.

| Algorithm | Number of Hidden Units | Number of weights | Learning Rate | % Success | Success After |
|---|---|---|---|---|---|
| RTRL | 3 | $\approx 170$ | 0.05 | "Some fraction" | 173,000 |
| RTRL | 12 | $\approx 494$ | 0.1 | "Some fraction" | 25,000 |
| ELM | 15 | $\approx 435$ | | 0 | >200,000 |
| RCC | 7–9 | $\approx 119$–198 | | 50 | 182,000 |
| Standard LSTM | 3 memory blocks, size 2 | 276 | 0.5 | 100 | 8440 |

Sources: RTRL: Results from Smith and Zipser (1989); ELM results from Cleeremans et al. (1989); RCC results from Fahlman (1991); and LSTM results from Hochreiter and Schmidhuber (1997).
Note: Weight numbers in the first four rows are estimates.

feed 10 test streams. Our performance measure is the average test stream size; 100,000 corresponds to a so-called perfect solution ($10^6$ successive correct predictions).

**4.2 Network Topology and Parameters.** The seven input units are fully connected to a hidden layer consisting of four memory blocks with 2 cells each (8 cells and 12 gates in total). The cell outputs are fully connected to the cell inputs, all gates, and the seven output units. The output units have additional "shortcut" connections from the input units (see Figure 3). All gates and output units are biased. Bias weights to input and output gates are initialized blockwise: $-0.5$ for the first block, $-1.0$ for the second, $-1.5$ for the third, and so forth. In this manner, cell states are initially close to zero; as training progresses, the biases become progressively less negative, allowing the serial activation of cells as active participants in the network computation. Forget gates are initialized with symmetric positive values: +0.5 for the first block, +1 for the second block, and so on. Precise bias initialization is not critical, though; other values work just as well. All other weights including the output bias are initialized randomly in the range $[-0.2, 0.2]$. There are 424 adjustable weights, which is comparable to the number used by LSTM in solving the ERG (see Table 1).

Weight changes are made after each input symbol presentation. At the beginning of each training stream, the learning rate $\alpha$ is initialized with 0.5. It either remains fixed or decays by a factor of 0.99 per time step (LSTM with $\alpha$-decay). Learning rate decay is well studied in statistical approximation theory and is also common in neural networks (e.g., Darken, 1995). We report results of exponential $\alpha$-decay (as specified above), but also tested several other variants (linear, $1/T$, $1/\sqrt{T}$) and found them all to work as well without extensive optimization of parameters.

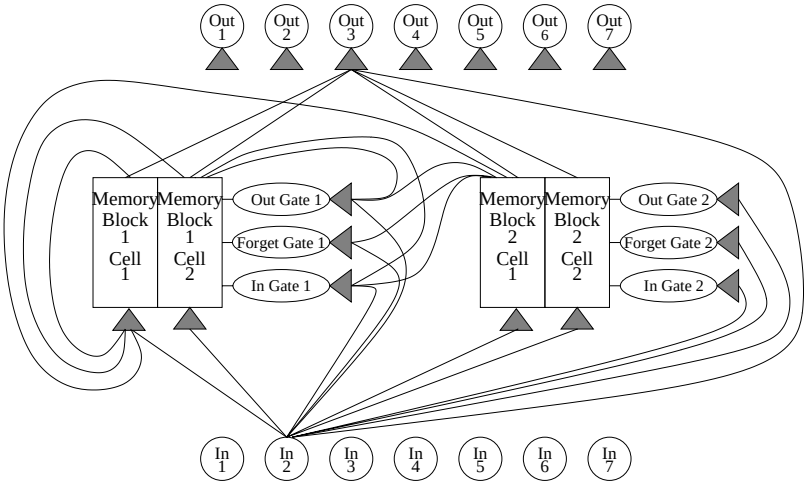


Figure 3: Three-layer LSTM topology with recurrence limited to the hidden layer consisting of four extended LSTM memory blocks (only two are shown) with two cells each. Only a limited subset of connections are shown.

**4.3 CERG Results.** Training was stopped after at most 30,000 training streams, each of which was ended when the first prediction error or the 100,000th successive input symbol occurred. Table 2 compares extended LSTM (with and without learning rate decay) to standard LSTM and an LSTM variant with decay of the internal cell state $s_c$ (with a self-recurrent weight $< 1$). Our results for standard LSTM with network activation resets (by an external teacher) at sequence ends are slightly better than those based on a different topology (Hochreiter & Schmidhuber, 1997). External resets (noncontinual case) allow LSTM to find excellent solutions in 74% of the trials, according to our stringent testing criterion. Standard LSTM fails, however, in the continual case. Internal state decay does not help much either (we tried various self-recurrent weight values and report only the best result). Extended LSTM with forget gates, however, can solve the continual problem.

A continually decreasing learning rate led to even better results but had no effect on the other algorithms. Different topologies may provide better results too. We did not attempt to optimize topology.

Can the network learn to recognize appropriate times for opening and closing its gates without using the information conveyed by the marker symbols **B** and **E**? To test this, we replaced all CERG subnets of the type $\xrightarrow{\mathbf{T\backslash P}} \bullet \xrightarrow{\mathbf{E}} \bullet \xrightarrow{\mathbf{B}} \bullet \xrightarrow{\mathbf{T\backslash P}}$ by $\xrightarrow{\mathbf{T\backslash P}} \bullet \xrightarrow{\mathbf{T\backslash P}}$. This makes the task more difficult, as the net now needs to keep track of sequences of numerous potentially confusing **T** and **P** symbols. But LSTM with forget gates (same topology)

Table 2: Continuous Embedded Reber Grammar (CERG).

| Algorithm | % Solutions[a] | % Good Solutions[b] | % Rest[c] |
|---|---|---|---|
| Standard LSTM with external reset | 74 (7441) | 0 ⟨−⟩ | 26 ⟨31⟩ |
| Standard LSTM | 0 (-) | 1 ⟨1166⟩ | 99 ⟨37⟩ |
| LSTM with state decay (0.9) | 0 (-) | 0 ⟨−⟩ | 100 ⟨56⟩ |
| LSTM with forget gates | 18 (18,889) | 29 ⟨39,171⟩ | 53 ⟨145⟩ |
| LSTM with forget gates and sequential $\alpha$ decay | 62 (14,087) | 6 ⟨68,464⟩ | 32 ⟨30⟩ |

[a]Percentage of "perfect" solutions (correct prediction of 10 streams of 100,000 symbols each). The number of training streams presented until a solution was reached is shown in parentheses.
[b]Percentage of solutions with an average stream length $> 1000$. The mean length of error-free prediction is given in angle brackets.
[c]Percentage of "bad" solutions with average stream length $\leq 1000$. The mean length of error-free prediction is given in angle brackets.
Notes: The results are averages over 100 independently trained networks. Other algorithms like BPTT are not included in the comparison, because they tend to fail even on the easier, noncontinual ERG.

was still able to find perfect solutions, although less frequently (sequential $\alpha$ decay was not applied).

**4.4 Analysis of the CERG Results.** How does extended LSTM solve the task on which standard LSTM fails? Section 2.1 already mentioned LSTM's problem of uncontrolled growth of the internal states. Figure 4 shows the evolution of the internal states $s_c$ during the presentation of a test stream. The internal states tend to grow linearly. At the starts of successive ERG strings, the network is in an increasingly active state. At some point (here,

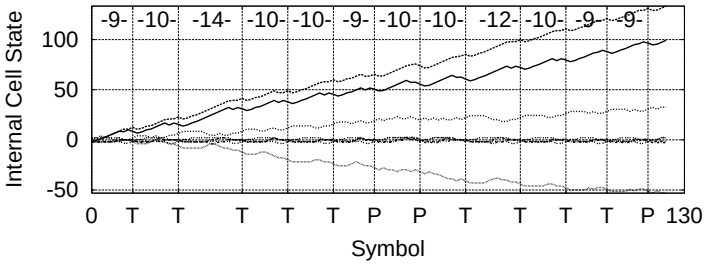


Figure 4: Evolution of standard LSTM's internal states $s_c$ during presentation of a test stream stopped at first prediction failure. Starts of new ERG strings are indicated by vertical lines labeled by the symbols (**P** or **T**) to be stored until the next string start.

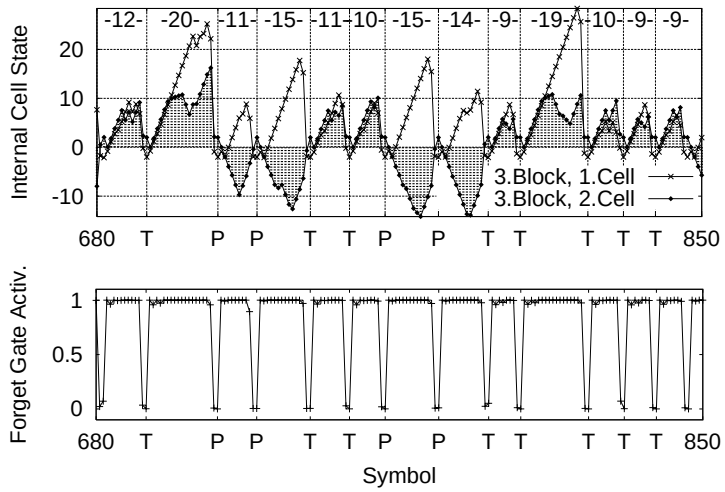


Figure 5: (Top) Internal states $s_c$ of the two cells of the self-resetting third memory block in an extended LSTM network during a test stream presentation. The figure shows 170 successive symbols taken from the longer sequence presented to a network that learned the CERG. Starts of new ERG strings are indicated by vertical lines labeled by the symbols **P** or **T**, to be stored until the next string start. (Bottom) Simultaneous forget gate activations of the same memory block.

after 13 successive strings), the high level of state activation leads to saturation of the cell outputs, and performance breaks down. Extended LSTM, however, learns to use the forget gates for resetting its state when necessary. Figure 5 (top) shows a typical internal state evolution after learning. We see that the third memory block resets its cells in synchrony with the starts of ERG strings. The internal states oscillate around zero; they never drift out of bounds as with standard LSTM (see Figure 4). It also becomes clear how the relevant information gets stored: the second cell of the third block stays negative, while the symbol **P** has to be stored, whereas a **T** is represented by a positive value. The third block's forget gate activations are plotted in Figure 5 (bottom). Most of the time, they are equal to 1.0, thus letting the memory cells retain their internal values. At the end of an ERG string, the forget gate's activation goes to zero, thus resetting cell states to zero.

Analyzing the behavior of the other memory blocks, we find that only the third is directly responsible for bridging ERG's longest time lag. Figure 6 plots values analogous to those in Figure 5 for the first memory block and its first cell. The first block's cell and forget gate show short-term behavior only (necessary for predicting the numerous short time lag events of the Reber

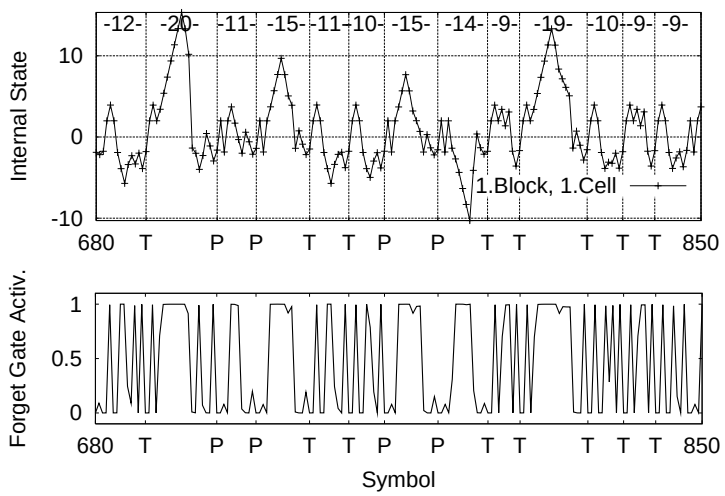


Figure 6: (Top) Extended LSTM's self-resetting states for the first cell in the first block. (Bottom) Forget gate activations of the first memory block.

grammar). The same is true for all other blocks except the third. Common to all memory blocks is that they learned to reset themselves in an appropriate fashion.

**4.5 Continual Noisy Temporal Order Problem.** Extended LSTM solves the CERG problem; standard LSTM does not. But can standard LSTM solve problems that extended LSTM cannot? We tested extended LSTM on one of the most difficult nonlinear long time lag tasks ever solved by an RNN: noisy temporal order (NTO) (task 6b taken from Hochreiter & Schmidhuber 1997).

*4.5.1 NTO.* The goal is to classify sequences of locally represented symbols. Each sequence starts with an $E$, ends with a $B$ (the "trigger symbol"), and otherwise consists of randomly chosen symbols from the set $\{a, b, c, d\}$ except for three elements at positions $t_1$, $t_2$, and $t_3$ that are either $X$ or $Y$. The sequence length is randomly chosen between 100 and 110, $t_1$ is randomly chosen between 10 and 20, $t_2$ is randomly chosen between 33 and 43, and $t_3$ is randomly chosen between 66 and 76. There are eight sequence classes, $Q, R, S, U, V, A, B, C$, which depend on the temporal order of the $X$s and $Y$s. The rules are: $X, X, X \rightarrow Q$; $X, X, Y \rightarrow R$; $X, Y, X \rightarrow S$; $X, Y, Y \rightarrow U$; $Y, X, X \rightarrow V$; $Y, X, Y \rightarrow A$; $Y, Y, X \rightarrow B$; $Y, Y, Y \rightarrow C$. Target signals occur only at the end of a sequence. The problem's minimal time lag size is 80. Forgetting is harmful only because all relevant information has to

Table 3: Continuous Noisy Temporal Order (CNTO).

| Algorithm | % Perfect Solution[a] | % Partial Solution[b] |
|---|---|---|
| Standard LSTM | 0 (-) | 100 ⟨4.6⟩ |
| LSTM with forget gates | 24 (18,077) | 76 ⟨12.2⟩ |
| LSTM with forget gates and sequential $\alpha$ decay | 37 (22,654) | 63 ⟨11.8⟩ |

[a]Percentage of perfect solutions (correct classification of 1000 successive NTO sequences in 10 test streams). In parentheses is the number of training streams presented.
[b]Percentage of solutions and average stream size (value in angular brackets) $\leq 100$.
Notes: All results are averages over 100 independently trained networks. Other algorithms (BPTT, RTRL, etc.) are not included in the comparison, because they fail even on the easier, noncontinual NTO.

be kept until the end of a sequence, after which the network is reset anyway.

We use the network topology described in section 4.2 with eight input and eight output units. Using a large bias (5.0) for the forget gates, extended LSTM solved the task as quickly as standard LSTM (recall that a high forget gate bias makes extended LSTM degenerate into standard LSTM). Using a moderate bias like the one used for CERG (1.0), extended LSTM took about three times longer on average, but did solve the problem. The slower learning speed results from the net's having to learn to remember everything and not to forget.

Generally we have not yet encountered a problem that LSTM solves while extended LSTM does not.

*4.5.2 CNTO.* Now we take the next obvious step and transform the NTO into a continual problem that does require forgetting, just as in section 4.1, by generating continual input streams consisting of concatenated NTO sequences. Processing such streams without intermediate resets, the network is required to learn to classify NTO sequences in an on-line fashion. Each input stream is stopped once the network makes an incorrect classification or 100 successive NTO sequences have been classified correctly. Learning and testing alternate; the performance measure is the average size of 10 test streams, measured by the number of their NTO sequences (each containing between 100 and 110 input symbols). Training is stopped after at most $10^5$ training streams.

*4.5.3 Results.* Table 3 summarizes the results. We observe that standard LSTM again fails to solve the continual problem. Extended LSTM with forget gates, however, can solve it. A continually decreasing learning rate ($\alpha$ decaying by a fraction of 0.9 after each NTO sequence in a stream) leads to slightly better results but is not necessary.

## 5 Conclusion

Continual input streams generally require occasional resets of the stream-processing network. Partial resets are also desirable for tasks with hierarchical decomposition. For instance, reoccurring subtasks should be solved by the same network module, which should be reset once the subtask is solved. Since typical real-world input streams are not a priori decomposed into training subsequences and typical sequential tasks are not a priori decomposed into appropriate subproblems, RNNs should be able to learn to achieve appropriate decompositions. Our novel forget gates naturally permit LSTM to learn local self-resets of memory contents that have become irrelevant.

Extended LSTM holds promise for any sequential processing task in which we suspect that a hierarchical decomposition may exist but do not know in advance what this decomposition is. The model has been successfully applied to the task of discriminating languages from very limited prosodic information (Cummins, Gers, & Schmidhuber, 1999) where there is no clear linguistic theory of hierarchical structure. Memory blocks equipped with forget gates may also be capable of developing into internal oscillators or timers, allowing the recognition and generation of hierarchical rhythmic patterns.

## Acknowledgments

This work was supported by SNF grant 2100-49'144.96, "Long Short-Term Memory," to J. S. Thanks to Nici Schraudolph for providing his fast exponentiation code (Schraudolph, 1999) employed to accelerate the computation of exponentials.

## Appendix: Summary of Extended LSTM in Pseudocode

**REPEAT** learning loop
**forward pass**
**net input** to hidden layer (self recurrent and from input)
**reset** all $net$ values with bias connection or to zero
input gates (1): $net_{in_j}(t) = \sum_m w_{in_j m}\ y^m(t-1)$
forget gates (6): $net_{\varphi_j}(t) = \sum_m w_{\varphi_j m}\ y^m(t-1)$
output gates (1): $net_{out_j}(t) = \sum_m w_{out_j m}\ y^m(t-1)$
cells (3): $net_{c_j^v}(t) = \sum_m w_{c_j^v m}\ y^m(t-1)$
**activations** in hidden layer
input gates (1): $y^{in_j}(t) = f_{in_j}(net_{in_j}(t))$
forget gates (6): $y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t))$
output gates (1): $y^{out_j}(t) = f_{out_j}(net_{out_j}(t))$
cells' internal states (7):
$s_{c_j^v}(0) = 0\ , s_{c_j^v}(t) = y^{\varphi_j}(t)\ \ s_{c_j^v}(t-1) + y^{in_j}(t)\ \ g(net_{c_j^v}(t))$
cells' activations (3): $y^{c_j^v}(t) = y^{out_j}(t)\ \ h(s_{c_j^v}(t))$
**net input** and **activations** of **output** units (5):
$net_k(t) = \sum_m w_{km}\ \ y^m(t-1)\ \ ,\quad y^k(t) = f_k(net_k(t))$
**derivatives** for input gates, forget gates and cells
variable $dS_{lm}^{jv}(t) := \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}}\ \ ,\ \ \bar{l} \in \{\varphi, in, c\}\ \ ,\ \ l \in \{\varphi_j, in_j, c_j^v\}$
input gates (15): $dS_{in,m}^{jv}(0) = 0,$
$dS_{in,m}^{jv}(t) = dS_{in,m}^{jv}(t-1)\ \ y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ \ f'_{in_j}(net_{in_j}(t))\ \ y^m(t-1)$
forget gates (16): $dS_{\varphi m}^{jv}(0) = 0,$
$dS_{\varphi m}^{jv}(t) = dS_{\varphi m}^{jv}(t-1)\ \ y^{\varphi_j}(t) + s_{c_j^v}(t-1)\ \ f'_{\varphi_j}(net_{\varphi_j}(t))\ \ y^m(t-1)$
cells (14): $dS_{cm}^{jv}(0) = 0,$
$dS_{cm}^{jv}(t) = dS_{cm}^{jv}(t-1)\ \ y^{\varphi_j}(t) + g'(net_{c_j^v}(t))\ \ y^{in_j}(t)\ \ y^m(t-1)$
**backward pass** if error injected
**errors** and $\delta$s
injection error (7): $e_k(t) := t^k(t) - y^k(t)$
output units (9): $\delta_k(t) = f'_k(net_k(t))\ \ e_k(t)$
output gates (10):
$\delta_{out_j}(t) = f'_{out_j}(net_{out_j}(t))\ \ \left(\sum_{v=1}^{S_j} h(s_{c_j^v}(t)) \sum_k w_{kc_j^v}\ \ \delta_k(t)\right)$
input gates, forget gates and cells (12):
$e_{s_{c_j^v}}(t) = y^{out_j}(t)\quad h'(s_{c_j^v}(t)) \left(\sum_k w_{kc_j^v}\ \ \delta_k(t)\right)$
**weight updates**
output units and output gates (8):
$\Delta w_{lm}(t) = \alpha\ \ \delta_l(t)\ \ y^m(t-1)\ \ ,\ \ l \in \{k, i, out\}$
input gates (19): $\Delta w_{in,m}(t) = \alpha\ \ \sum_{v=1}^{S_j}\ \ e_{s_{c_j^v}}(t)\quad dS_{in,m}^{jv}(t)$
forget gates (19): $\Delta w_{\varphi m}(t) = \alpha\ \ \sum_{v=1}^{S_j}\ \ e_{s_{c_j^v}}(t)\quad dS_{\varphi m}^{jv}(t)$
cells (18): $\Delta w_{c_j^v m}(t) = \alpha\ \ e_{s_{c_j^v}}(t)\ \ dS_{cm}^{jv}(t)$
**UNTIL** error stopping criterion

## References


Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks, 5*(2), 157–166.

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. L. (1989). Finite-state automata and simple recurrent networks. *Neural Computation, 1*, 372–381.

Cummins, F., Gers, F., & Schmidhuber, J. (1999). Language identification from prosody without explicit features. In *Proceedings of EUROSPEECH'99* (Vol. 1, pp. 371–374).

Darken, C. (1995). Stochastic approximation and neural network learning. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks* (pp. 941–944). Cambridge, MA: MIT Press.

Doya, K., & Yoshizawa, S. (1989). Adaptive neural oscillator using continuous-time backpropagation learning. *Neural Networks, 2*(5), 375–385.

Fahlman, S. E. (1991). The recurrent cascade-correlation learning algorithm. In R. P. Lippmann, J. E. Moody, & D. S. Touretzky (Eds.), *Advances in neural information processing systems, 3* (pp. 190–196). San Mateo, CA: Morgan Kaufmann.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). *Learning to forget: Continual prediction with LSTM* (Tech. Rep. No. IDSIA-01-99). Lugano, Switzerland: IDSIA.

Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Technische Universität München*. Available online at www7.informatik.tu-muenchen.de/~hochreit.

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

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*. Hillsdale, NJ: Erlbaum.

Lin, T., Horne, B. G., Tiño, P., & Giles, C. L. (1996). Learning long-term dependencies in NARX recurrent neural networks. *IEEE Transactions on Neural Networks, 7*(6), 1329–1338.

Mozer, M. C. (1989). A focused backpropagation algorithm for temporal pattern processing. *Complex Systems, 3*, 349–381.

Pearlmutter, B. A. (1995). Gradient calculation for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks, 6*(5), 1212–1228.

Robinson, A. J., & Fallside, F. (1987). *The utility driven dynamic error propagation network*. (Tech. Rep. No. CUED/F-INFENG/TR.1). Cambridge: Cambridge University Engineering Department.

Schmidhuber, J. (1989). The neural bucket brigade: A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science, 1*(4), 403–412.

Schmidhuber, J. (1992). A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation, 4*(2), 243–248.

Schraudolph, N. (1999). A fast, compact approximation of the exponential function. *Neural Computationx, 11*(4), 853–862.

Smith, A. W., & Zipser, D. (1989). Learning sequential structures with the real-time recurrent learning algorithm. *International Journal of Neural Systems, 1*(2), 125–131.

Waibel, A. (1989). Modular construction of time-delay neural networks for speech recognition. *Neural Computation, 1*(1), 39–46.

Werbos, P. J. (1988). Generalisation of backpropagation with application to a recurrent gas market model. *Neural Networks, 1*, 339–356.

Williams, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation, 2*(4), 490–501.

Williams, R. J., & Zipser, D. (1992). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin & D. E. Rumelhart (Eds.), *Back-propagation: Theory, architectures and applications*. Hillsdale, NJ: Erlbaum.

Received March 3, 1999; accepted November 7, 1999.