Learning to Forget: Continual Prediction with LSTM

Article *in* Neural Computation · October 2000

DOI: 10.1162/089976600300015015 · Source: PubMed

3 authors, including:

Felix Gers
Beuth Hochschule für Technik Berlin
63 PUBLICATIONS **7,570** CITATIONS

SEE PROFILE

Fred Cummins
University College Dublin
90 PUBLICATIONS **5,167** CITATIONS

SEE PROFILE

Learning to Forget: Continual Prediction with LSTM

Technical Report IDSIA-01-99
January, 1999

Felix A. Gers felix@idsia.ch
Jürgen Schmidhuber juergen@idsia.ch
Fred Cummins fred@idsia.ch

IDSIA, Corso Elvezia 36
6900 Lugano, Switzerland
www.idsia.ch

Abstract

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 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 and Zipser, 1992; Werbos, 1988) or "Real-Time Recurrent Learning" (Robinson and Fallside, 1987; Williams and Zipser, 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 quickly either vanishes or blows up (Hochreiter and Schmidhuber, 1997; Bengio et al., 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.

A recent model, *"Long Short-Term Memory"* (LSTM), (Hochreiter and 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 carrousels" (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 comparison of LSTM to alternative approaches.

In this paper, however, we will show that even LSTM fails to learn to correctly process certain very long or continual time series 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 paper will present a remedy.

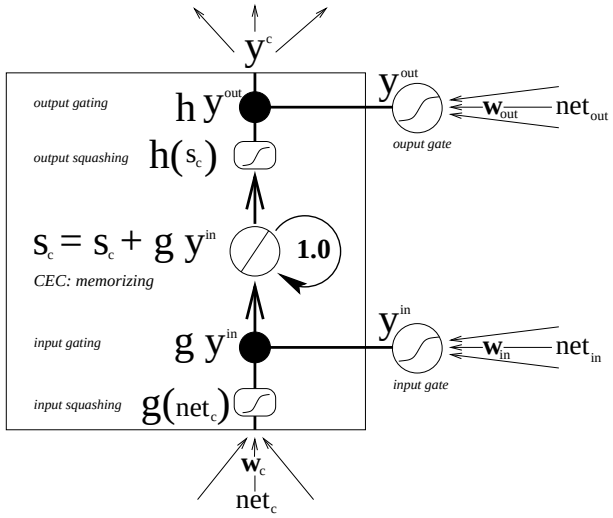

Figure 1: The standard LSTM cell has a linear unit with a recurrent self-connection with weight 1.0 (CEC). Input and output gate regulate read and write access to the cell whose state is denoted s_c. The function g squashes the cell's input; h squashes the cell's output (see text for details).

While we present a specific solution to the problem of forgetting in LSTM networks, we recognize that *any* training procedure for RNNs which 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 which is sufficiently powerful to have encountered this problem.

Outline. 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 any more. 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 a more complex, continual task, 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 which 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 CEC's solve the vanishing error problem: in the absence of new input or error signals to the cell, the CEC's local error back flow remains constant, neither growing nor decaying. The CEC is protected from both forward flowing activation and backward flowing error by the input and 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:

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

$$net_{in_j}(t) \quad = \quad \sum_m w_{in_j m} \; y^m(t-1) \; ; \quad y^{in_j}(t) = f_{in_j}(net_{in_j}(t)) \quad . \tag{2}$$

Throughout this paper 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 the gates, f is a logistic sigmoid (with range $[0,1]$):

$$f(x) = \frac{1}{1+\mathrm{e}^{-x}} \quad . \tag{3}$$

The input to the cell itself is

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m} \; y^m(t-1) \quad , \tag{4}$$

which is is squashed by g, a centered logistic sigmoid function with range $[-2,2]$:

$$g(x) = \frac{4}{1+\mathrm{e}^{-x}} - 2 \quad . \tag{5}$$

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

$$s_{c_j^v}(0) = 0 \; ; \quad s_{c_j^v}(t) = s_{c_j^v}(t-1) + y^{in_j}(t) \; g(net_{c_j^v}(t)) \quad \text{for } t > 0 \; . \tag{6}$$

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}:

$$y^{c_j^v}(t) = y^{out_j}(t) \; h(s_{c_j^v}(t)) \quad . \tag{7}$$

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

$$h(x) = \frac{2}{1+\mathrm{e}^{-x}} - 1 \quad . \tag{8}$$

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:

$$net_k(t) = \sum_m w_{km} \; y^m(t-1) \; , \quad y^k(t) = f_k(net_k(t)) \quad , \tag{9}$$

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 (3). This concludes standard LSTM's forward pass. All equations except for equation (6) will remain valid for extended LSTM with forget gates.

Learning. See Hochreiter & Schmidhuber (1997) for details of standard LSTM's backward pass. Essentially, as in truncated 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. In essence, once an error signal arrives at a memory cell output, it gets scaled by the output gate and the output nonlinearity h; then it 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 will be 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, e.g., at the beginnings of new input sequences (whose starts, however, are not explicitly indicated by a teacher). Saturation will (a) make h's derivative vanish, thus blocking incoming errors, and (b) 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. This problem did not arise in the experiments reported in Hochreiter & Schmidhuber (1997) because cell states were explicitly reset to zero before the start of each new sequence.

In principle, the network might use other memory cells to reset cells containing useless information by feeding them appropriate, inversely signed inputs. This does not work well in practice, however, because the precise values needed to reset the cells state are hard to learn: This would necessitate (among other things) computing the inverse of the output squashing function h. In general the network will not automatically reset itself to a neutral state once a new training sequence starts.

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 et al., 1996), which depend on *a priori* knowledge of typical time lag sizes?

Weight decay does not work. It could only slow down the growth of cell states indirectly by decreasing the overall activity in the network. We tested several weight decay algorithms (Hinton, 1986), (Weigend et al., 1991) without any encouraging results. 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 and Shuji, 1989) the internal states s_c by resetting them once a new training sequence starts. But this requires an external teacher that 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 are adaptive "forget gates" designed to learn to reset memory blocks once their contents are out of date and hence useless. By resets we do not only mean 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 (Figure 1) by the multiplicative forget gate activation y^{φ}. See Figure 2.

3.1 Forward Pass of Extended LSTM with Forget Gates

The forget gate activation y^{φ} is calculated like the activations of the other gates—see equations (1) and (2):

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m}\ y^m(t-1)\ ; \quad y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)) \quad . \tag{10}$$

Here net_{φ_j} is the input from the network to the forget gate. We use the logistic sigmoid (3) as squashing function, hence the forget gate's activation y^{φ} ranges between 0 and 1. Its output

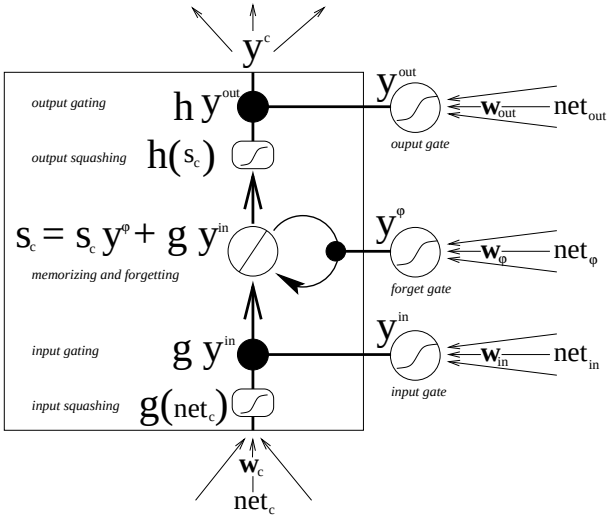

Figure 2: Memory block with only one cell for the extended LSTM. A multiplicative forget gate can reset the cell's inner state s_c.

becomes the weight of the self recurrent connection of the internal state s_c in equation (6). The revised update equation for s_c in the extended LSTM algorithm is:

$$\begin{aligned} s_{c_j^v}(0) &= 0 \,, \\ s_{c_j^v}(t) &= y^{\varphi_j}(t)\; s_{c_j^v}(t-1) + y^{in_j}(t)\; g(net_{c_j^v}(t)) \quad \text{for } t > 0 \,. \end{aligned} \tag{11}$$

Extended LSTM's full forward pass is obtained by adding equations (10) to those in Section 2 and replacing equation (6) by (11).

Bias weights for LSTM gates are initialized with negative values for input and output gates (see Hochreiter & Schmidhuber 1997) for details), positive values for forget gates. This implies—compare equations (10) and (11)—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 back propagation through time (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 of cells tends to decay exponentially anyway (Hochreiter, 1991). In the equations below, $\stackrel{tr}{=}$ will indicate where we use error truncation and, for simplicity, unless otherwise indicated, we assume only a single cell per block.
We start with the usual squared error objective function based on targets t^k:

$$E(t) = \frac{1}{2} \sum_k e_k(t)^2 \;\; ; \; e_k(t) := t^k(t) - y^k(t) \;\; , \tag{12}$$

where e_k denotes the externally injected error. We minimize E via gradient descent by adding

weight changes Δw_{lm} to the weights w_{lm} (from unit m to unit l) using learning rate α:

$$\begin{aligned}
\Delta w_{lm}(t) &= -\alpha \frac{\partial E(t)}{\partial w_{lm}} = -\alpha \frac{\partial E(t)}{\partial y^k(t)} \frac{\partial y^k(t)}{\partial w_{lm}} = \alpha \sum_k e_k(t) \frac{\partial y^k(t)}{\partial w_{lm}} = \\
&\stackrel{tr}{=} \alpha \sum_k e_k(t) \frac{\partial y^k(t)}{\partial y^l(t)} \frac{\partial y^l(t)}{\partial net_l(t)} \underbrace{\frac{\partial net_l(t)}{\partial w_{lm}}}_{=y^m(t-1)} \\
\Delta w_{lm}(t) &= \alpha \underbrace{\frac{\partial y^l(t)}{\partial net_l(t)} \left(\sum_k \frac{\partial y^k(t)}{\partial y^l(t)} e_k(t) \right)}_{=:\ \delta_l(t)} y^m(t-1) \quad . \qquad (13)
\end{aligned}$$

For an arbitrary output unit ($l = k$) the sum in (13) vanishes. By differentiating equation (9) we obtain the usual back-propagation weight changes for the output units:

$$\frac{\partial y^k(t)}{\partial net_k(t)} = f_k'(net_k(t)) \implies \delta_k(t) = f_k'(net_k(t))\ e_k(t) \quad . \qquad (14)$$

For all hidden units directly connected to the output layer, we have $\frac{\partial y^k(t)}{\partial y^l(t)} e_k(t) = w_{kl}\delta_k(t)$. To compute the weight changes for the output gates $\Delta w_{out_j m}$ we set ($l = out$) in (13). The resulting terms can be determined by differentiating equations (1), (7) and (9):

$$\begin{aligned}
\frac{\partial y^{out_j}(t)}{\partial net_{out_j}(t)} &= f_{out_j}'(net_{out_j}(t)) \quad , \\
\frac{\partial y^k(t)}{\partial y^{out_j}(t)}\ e_k(t) &= \frac{\partial y^{c_j^v}}{\partial y^{out_j}(t)} \underbrace{\frac{\partial y^k(t)}{\partial y^{c_j^v}(t)}\ e_k(t)}_{=w_{kc_j^v}\ \delta_k(t)} = h(s_{c_j^v}(t))\ w_{kc_j^v}\ \delta_k(t) \quad .
\end{aligned}$$

Inserting both terms in equation (13) gives the contribution of the block's v-th cell to δ_{out_j}:

$$\delta_{out_j}^v(t) = f_{out_j}'(net_{out_j}(t))\ h(s_{c_j^v}(t)) \left(\sum_k w_{kc_j^v}\ \delta_k(t) \right) \quad .$$

As 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 δ_{out_j} of the j-th memory block (with S_j cells):

$$\delta_{out_j}(t) = f_{out_j}'(net_{out_j}(t)) \left(\sum_{v=1}^{S_j} h(s_{c_j^v}(t)) \sum_k w_{kc_j^v}\ \delta_k(t) \right) \quad . \qquad (15)$$

Now we are done with all weights to output units and output gates of memory blocks. Equations (13), (14)and (15) define their weight changes. Until now everything 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 (13):

$$\Delta w_{lm}(t) = -\alpha \frac{\partial E(t)}{\partial w_{lm}} \stackrel{tr}{=} -\alpha \underbrace{\frac{\partial E(t)}{\partial s_{c_j^v}(t)}}_{=:\ -e_{s_{c_j^v}}(t)} \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} = \alpha\ e_{s_{c_j^v}}(t) \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \quad . \qquad (16)$$

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 (13) and obtain

$$e_{s_{c_j^v}}(t) := -\frac{\partial E(t)}{\partial s_{c_j^v}(t)} \stackrel{tr}{=} -\frac{\partial E(t)}{\partial y^k(t)} \frac{\partial y^k(t)}{\partial y^{c_j^v}(t)} \frac{\partial y^{c_j^v}(t)}{\partial s_{c_j^v}(t)} =$$

$$e_{s_{c_j^v}}(t) = \frac{\partial y^{c_j^v}}{\partial s_{c_j^v}(t)} \left(\sum_k \underbrace{\frac{\partial y^k(t)}{\partial y^{c_j^v}(t)} e_k(t)}_{=w_{c_j^v l}\ \delta_k(t)} \right) .$$

Differentiating the forward pass equation (7) we obtain:

$$\frac{\partial y^{c_j^v}}{\partial s_{c_j^v}(t)} = y^{out_j}(t)\ h'(s_{c_j^v}(t)) .$$

Substituting this term in the equation for $e_{s_{c_j^v}}$:

$$\Longrightarrow e_{s_{c_j^v}}(t) = y^{out_j}(t)\ h'(s_{c_j^v}(t)) \left(\sum_k w_{kc_j^v}\ \delta_k(t) \right) . \tag{17}$$

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 (16) we differentiate equation (11) and obtain a sum of four terms.

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} = \underbrace{\frac{\partial s_{c_j^v}(t-1)}{\partial w_{lm}} y^{\varphi_j}(t)}_{\neq 0 \text{ for all } l \in \{\varphi, in, c_j^v\}} + \underbrace{y^{in_j}(t) \frac{\partial g(net_{c_j^v}(t))}{\partial w_{lm}}}_{\neq 0 \text{ for } l = c_j^v \text{ (cell)}}$$
$$+ \underbrace{g(net_{c_j^v}(t)) \frac{\partial y^{in_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = in \text{ (input gate)}} + \underbrace{s_{c_j^v}(t-1) \frac{\partial y^{\varphi_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = \varphi \text{ (forget gate)}} . \tag{18}$$

The first term is nonzero for weights w_{lm} with $l \in \{\varphi, in, c_j^v\}$, i.e. all weights to the cell itself, to the input gate and to the forget gate. The next three terms are nonzero for only one value of l. Differentiating the forward pass equations (4), (2) and (10) for g, y^{in}, and y^{φ} we can substitute the unresolved partials and split the expression on the right hand side of (18) into three separate equations for the cell ($l = c_j^v$), the input gate ($l = in$) and the forget gate ($l = \varphi$):

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} y^{\varphi_j}(t) + g'(net_{c_j^v}(t))\ y^{in_j}(t)\ y^m(t-1) , \tag{19}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ f'_{in_j}(net_{in_j}(t))\ y^m(t-1) , \tag{20}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} y^{\varphi_j}(t) + h(s_{c_j^v}(t))\ f'_{\varphi_j}(net_{\varphi_j}(t))\ y^m(t-1) . \tag{21}$$

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

$$\frac{\partial s_{c_j^v}(t=0)}{\partial w_{lm}} = 0 \quad \text{for } l \in \{\varphi, in, c_j^v\} . \tag{22}$$

Note that the recursions in all equations (19)-(21) 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 (16) and calculate the corresponding weight updates, with the internal state error $e_{s_{c_j^v}}(t)$ given by equation (17). 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 $\Delta w_{c_j^v m}$ only depend on the partials of this cell's own state:

$$\Delta w_{c_j^v m}(t) = \alpha \; e_{s_{c_j^v}}(t) \; \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \quad . \tag{23}$$

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

$$\Delta w_{lm}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \quad \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \quad \text{for} \;\; l \in \{\varphi, in\} \quad . \tag{24}$$

The equations necessary to implement the backward pass are (13), (14)(15), (17), (19), (20), (21), (22), (23) and (24).

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 6). 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:

$$\begin{aligned} W_c &= \underbrace{(B \cdot (S + 2S + 1)) \cdot (B \cdot (S + 2)) + B \cdot (2S + 1)}_{\text{recurrent connections and bias}} \\ &+ \underbrace{(B \cdot S + 1) \cdot K}_{\text{to output}} + \underbrace{(B \cdot (S + 2S + 1)) \cdot I}_{\text{from input}} \quad . \end{aligned} \tag{25}$$

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 (19), (20) and (21) 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 (Schmidhuber, 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, e.g., Smith and Zipser (1989), Cleeremans et al. (1989), Fahlman (1991), Hochreiter & Schmidhuber (1997). Consider Figure 3.

ERG. The traditional method starts at the leftmost node of the ERG graph, and sequentially generates finite symbol strings (beginning with the empty string) by following edges, and appending the associated symbols to the current string until the rightmost node is reached. Edges are chosen randomly if there is a choice (probability = 0.5).

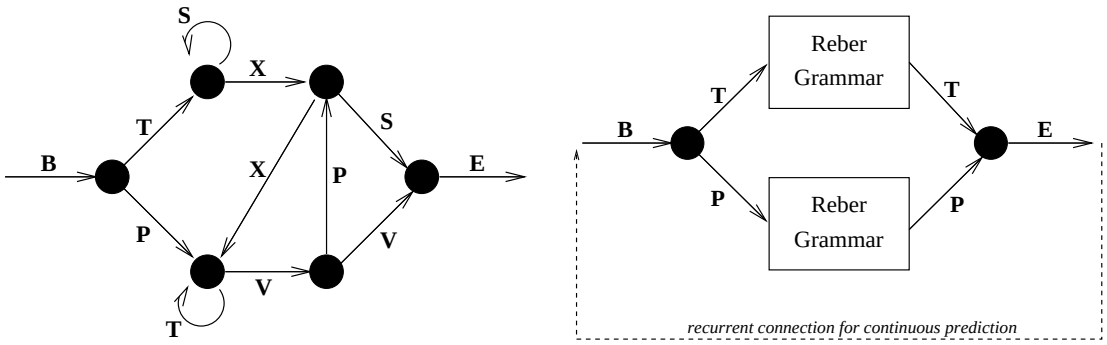

Figure 3: Transition diagrams for standard (left) and embedded (right) Reber grammars. The dashed line indicates the continual variant.

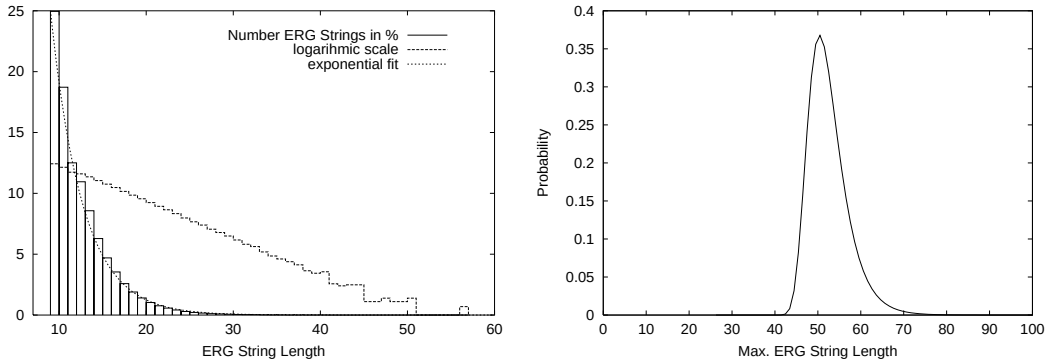

Figure 4: Left: histogram of 10^6 random samples of ERG string sizes. Right: Joint probability that an ERG string of a given size occurs and is the longest among 80000.

Input and target patterns are represented by 7 dimensional binary vectors, each component standing for one of the 7 possible symbols. Hence the network has 7 input units and 7 output units. The task is to read strings, one symbol at a time, and to continually predict 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 7 output units is below 0.49 (error signals occur at every time step).

To correctly predict 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 9 (at the limit of what standard recurrent networks can manage); time lags have no upper bound though. To provide an idea of the time lag distribution, Figure 4 (left) shows a histogram computed from sampled data.

ERG string probabilities decrease exponentially with ERG string size (see logarithmic plot of the exponential fit in Figure 4 (left)). From the data shown in the histogram of Figure 4 we calculate the probability $p(l)$ of sampling a string of size l:

$$p(l) \approx b\ \mathrm{e}^{-a(l-o)}\ ,\ a = b = \frac{3}{11}\ ,\ o = 9\ ,\ \text{for}\ \ l \geq 9\ \ \text{else}\ p(l) = 0\ \ .$$

Here o is an offset expressing the minimum string size of 9. To compute the normalized probability $P(L)$ of sampling a string of size $l <= L$ we integrate over $p(l)$:

$$P(L) = \int_o^L p(l)\ dl = \frac{b}{a}\left(1 - \mathrm{e}^{-a(L-o)}\right) = 1 - \mathrm{e}^{-a(L-o)}\ \ .$$

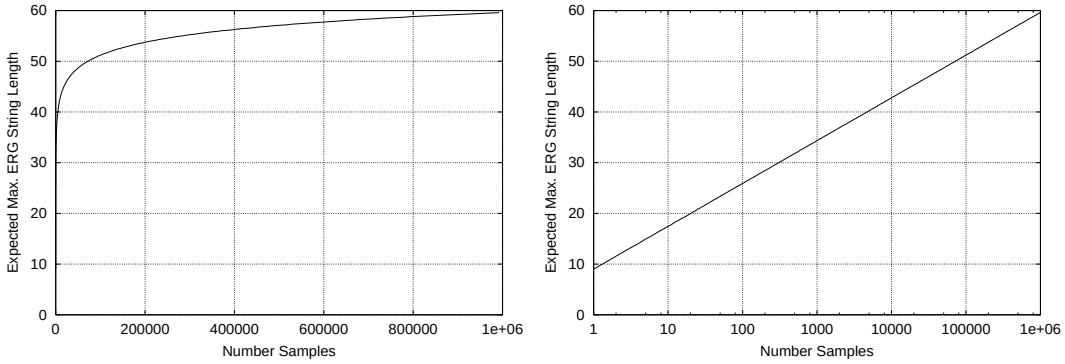

Figure 5: Left: number of embedded Reber strings plotted against lower bounds of expected maximal string size. Right: logarithmic x-axis.

Algo-rithm	# hidden units	#weights	learning rate	% of success	success after
RTRL	3	≈ 170	0.05	"some fraction"	173,000
RTRL	12	≈ 494	0.1	"some fraction"	25,000
ELM	15	≈ 435		0	>200,000
RCC	7-9	$\approx$ 119-198		50	182,000
Std. LSTM	3bl., size 2	276	0.5	100	8,440

Table 1: Standard embedded Reber grammar (ERG): percentage of successful trials and number of sequence presentations until success for RTRL (results taken from Smith and Zipser 1989), "Elman net trained by Elman's procedure" (results taken from Cleeremans et al. 1989), "Recurrent Cascade-Correlation" (results taken from Fahlman 1991) and LSTM (results taken from Hochreiter and Schmidhuber 1997). Weight numbers in the first 4 rows are estimates.

Solving $P(\bar{l}) \stackrel{!}{=} 1 - P(\bar{l})$ we find the expected ERG string size:

$$\bar{l} = o - \frac{1}{a} \ln(\frac{1}{2}) \approx 11.54 \quad . \tag{26}$$

Given a set of N ERG strings, what is the expected maximal time lag $\bar{\tau}(N)$ during which the network has to store information before using it? We derive a lower bound $P_N(\tau)$ for the probability that a set of N ERG strings contains a string of size $\geq \tau$, assuming a sample of $N-1$ strings of size $\leq \tau$ and one of size $\geq \tau$:

$$P_N(\tau) = N \cdot P(\tau)^N \cdot (1 - P(\tau)) \quad .$$

Figure 4 (right) plots P_N for $N = 80000$. The x-value of the distribution maximum is a lower bound for $\bar{\tau}(N = 80000)$. Figure 5 plots N against the lower bound of $\bar{\tau}(N)$. $\bar{\tau}(N)$ grows logarithmically with N.

Table 1 summarizes 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.

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

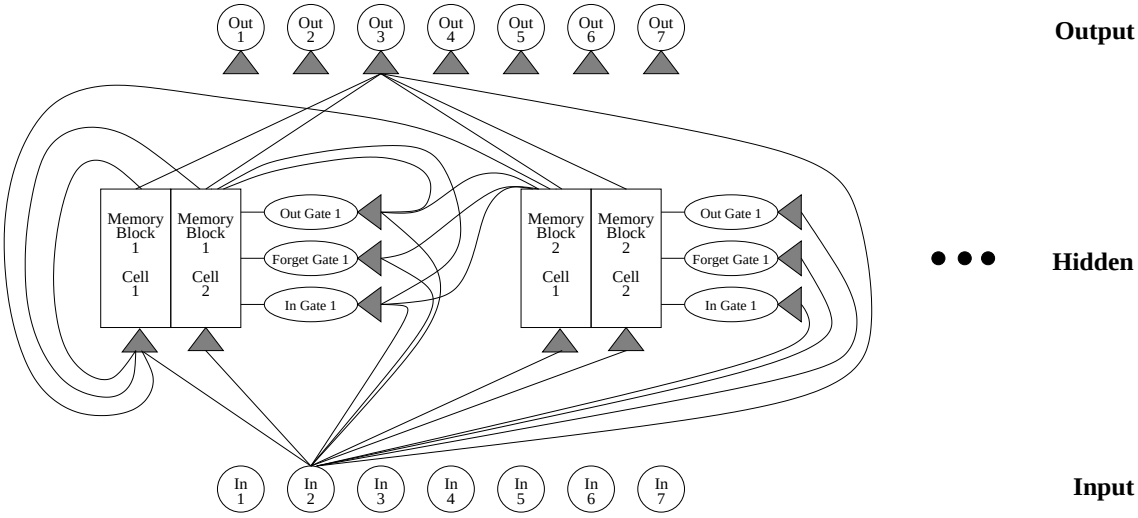

Figure 6: Three layer LSTM topology with recurrence limited to the hidden layer consisting of four extended LSTM memory blocks (only two shown) with two cells each. Only a limited subset of connections are shown.

stream we freeze the weights and 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).

Given the average ERG string size of 12 (equation (26)), a test with 10 times 10^5 symbols involves more than $80,000$ strings, that is, the expected maximal ERG string size encountered during a test of a perfect solution exceeds 50. To see this, consider Figure 4 (right) and the logarithmic plot of Figure 5 (right).

4.2 Network Topology and Parameters

The 7 input units are fully connected to a hidden layer consisting of 4 memory blocks with 2 cells each (8 cells and 12 gates in total). The cell outputs are fully connected to the cell inputs, to all gates, and to the 7 output units. The output units have additional "shortcut" connections to the input units (see Figure 6). All gates and output units are biased. Bias weights to in- 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, and, 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, etc. 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 375 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. The learning rate α is initialized with 0.5. It either remains fixed or slowly decays (Section 4.5).

4.3 CERG Results

Training was stopped after at most 30000 training streams, each of which was ended when the first prediction error or the 100000th successive input symbol occurred. Table 2 compares extended LSTM 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 external resets are slightly better than those based on a different topology (Hochreiter and Schmidhuber, 1997). External resets (non-continual) case allow LSTM to find excellent solutions in 74% of the trials, according to our

Algorithm	%Solutions	%Good Results	%Rest
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 (18889)	29 ⟨39171⟩	53 ⟨145⟩
LSTM with Forget Gates and α decay	62 (14087)	6 ⟨68464⟩	32 ⟨30⟩

Table 2: Continuous Embedded Reber Grammar (CERG): Column "%Solutions": Percentage of "perfect" solutions (correct prediction of 10 streams of 100,000 symbols each), in parenthesis the number of training streams presented until solution was reached. Column "Good Results": Percentage of solutions with an average stream length > 1000 (mean length of error free prediction is given in angle brackets). Column "Rest": percentage of "bad" solutions with average stream length ≤ 1000 (mean length of error free prediction is given in angle brackets). 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, non-continual ERG.

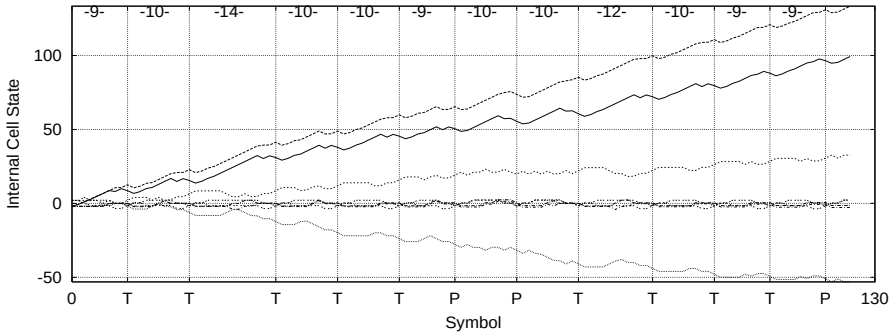

Figure 7: Evolution of standard LSTM's internal states s_c during presentation of a test stream stopped at first prediction failure. Starts of new ERG strings are indicated by vertical lines labeled by the symbols (**P** or **T**) to be stored until the next string start.

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 (discussion below). Different topologies may provide better results, too—we did not attempt to optimize topology.

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 7 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 after 13 successive strings), the high level of state activation leads to saturation of the cell outputs, and performance breaks down. Extended LSTM, however,

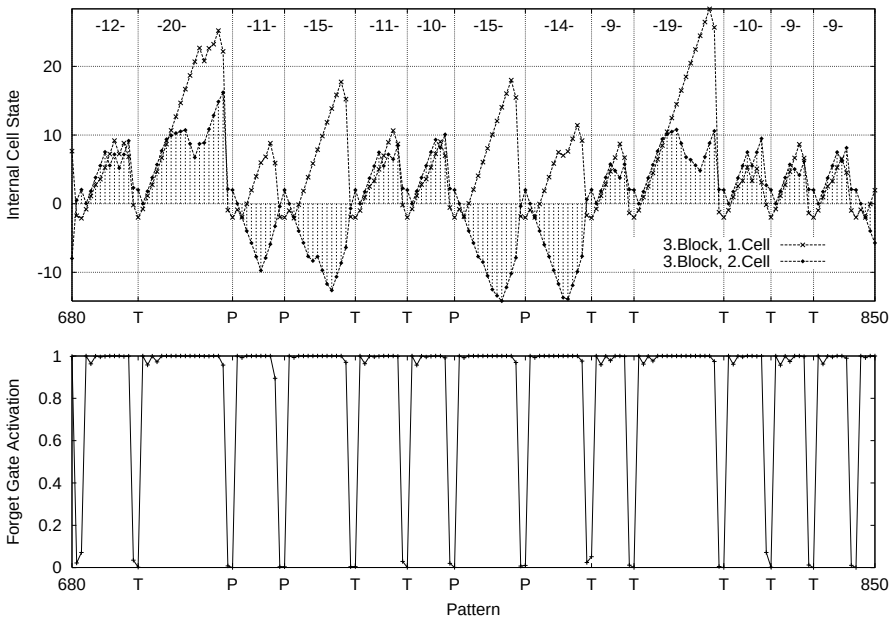

Figure 8: Top: Internal states s_c of the two cells of the self-resetting third memory block in an extended LSTM network during a test stream presentation. The figure shows 170 successive symbols taken from the longer sequence presented to a network that learned the CERG. Starts of new ERG strings are 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.

learns to use the forget gates for resetting its state when necessary. Figure 8 (top half) 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 bound as with standard LSTM (Figure 7). 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 8 (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 9 plots values analogous to those in Figure 8 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 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 Decreasing Learning Rate

Most on-line learning gradient-based neural network partially forget "long-term memory" (LTM) embodied by their slowly changing weights: memory traces of recent training exemplars tend to overwrite those of old ones. This well-known form of forgetting (Sutton, 1988) is different from the main topic of this paper, which focuses on learning (via weight changes) to forget short-term memory (STM) contents stored as quickly changing activations[1].

Still, LSTM is affected by the general problem. This suggests that it might profit from a remedy used in many previous on-line algorithms: simply let the learning rate α decay over time.

[1] Due to its ability to store activation-based memory for thousands of time steps, however, time lags bridged by LSTM's STM are in a range traditionally reserved for LTM (this being the main reason for LSTM's name).

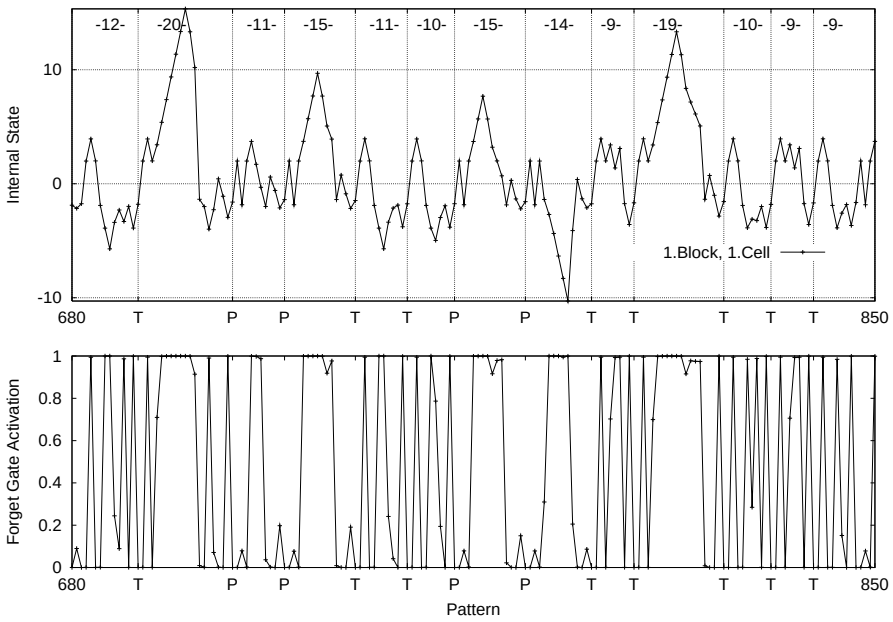

Figure 9: Top: Extended LSTM's self-resetting states for the first cell in the first block. Bottom: forget gate activations of the first memory block.

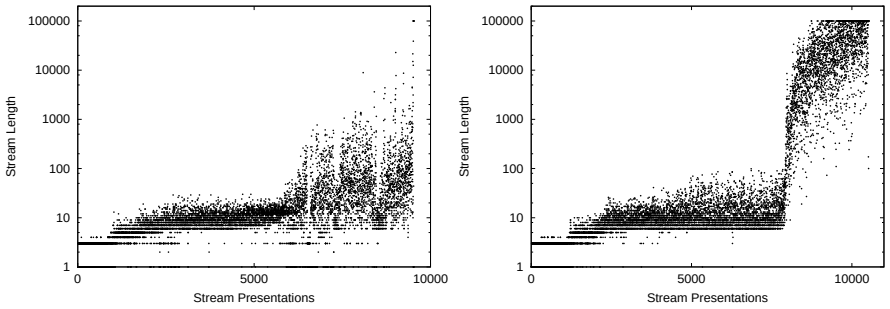

Figure 10: CERG: Training performance evolution of LSTM with forget gates (logarithmic y-axis). Every dot represents the length of one training stream, stopped after a prediction error occurred. Left: With constant learning rate $\alpha = 0.5$. Right: With exponential α-decay (at each time step, α decays by a fraction of 0.99).

In the CERG experiment an exponential α-decay helped to stabilize training and to improve the overall results. We initialized α with 0.5 and let it decay exponentially with a time constant of 100 input symbols by setting $\alpha := \gamma\alpha$ after each time step, where $\gamma = 0.99$, which led to good results (Table 2). Alternative $\gamma \in (0.9, 0.999)$ and initializations $\alpha \in (0.1, 1)$ worked as well—these values are not critical. Extended LSTM's different learning behavior with α-decay is illustrated by two typical learning curves shown in Figure 10. Without learning rate decay the network's performance tends to break down several times before arriving at the best solution of a run. α-decay, however, typically does find a perfect solution (10 times 100,000 successive symbols) without any intermediate break-downs.

Although α-decay does stabilize extended LSTM, it is not necessary. Perfect solutions can be obtained, for example, by simply reinitializing the network once it has not found a perfect solution within, say, 20000 training streams.

Algorithm	%Perfect Solutions	%Partial Solutions
Standard LSTM	0 ($> 10^5$)	100 $\langle 4.6 \rangle$
LSTM with Forget Gates	24 (74977)	76 $\langle 12.2 \rangle$
LSTM with Forget Gates and sequential α decay	37 (79354)	63 $\langle 11.8 \rangle$

Table 3: Continuous Noisy Temporal Order (CNTO): Column "%Perfect Solutions": Percentage of "perfect" solutions (correct classification of 1000 successive NTO sequences in 10 test streams); in parentheses: number of training streams presented. Column "%Partial Solutions": percentage of solutions and average stream size (value in angular brackets) ≤ 100. 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.

4.6 Continual Noisy Temporal Order Problem

Extended LSTM solves the CERG problem while standard LSTM does not. But can standard LSTM solve problems which 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).

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 8 sequence classes Q, R, S, U, V, A, B, C which depend on the temporal order of the Xs and Ys. 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 only harmful as all relevant information has to 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 8 input and 8 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 having to learn to remember everything and not to forget.
Generally speaking, we have not yet encountered a problem that LSTM solves while extended LSTM does not.

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 online 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.

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 (α 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, re-occurring 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 since 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.

LSTM's gates (and forget gates in particular) provide an example of local, efficient information processing through adaptive, multiplicative units, which are, due to their biological plausibility, attracting attention in the field of neuroscience (O'Reilly et al., 1999, in press).

Future work. 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. We will apply the model to on-line speech processing, especially in the domain of prosody, where linguistic theory does not yet have a clear 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.

Acknowledgment

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

6 Summary of Extended LSTM in Pseudo-code

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 (10): $net_{\varphi_j}(t) = \sum_m w_{\varphi_j m}\ y^m(t-1)$
    output gates (1): $net_{out_j}(t) = \sum_m w_{out_j m}\ y^m(t-1)$
    cells (4): $net_{c_j^v}(t) = \sum_m w_{c_j^v m}\ y^m(t-1)$
  activations in hidden layer
    input gates (1): $y^{in_j}(t) = f_{in_j}(net_{in_j}(t))$
    forget gates (10): $y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t))$
    output gates (1): $y^{out_j}(t) = f_{out_j}(net_{out_j}(t))$
    cells' internal states (11):
    $s_{c_j^v}(0) = 0\ , s_{c_j^v}(t) = y^{\varphi_j}(t)\ s_{c_j^v}(t-1) + y^{in_j}(t)\ g(net_{c_j^v}(t))$
    cells' activations (6): $y^{c_j^v}(t) = y^{out_j}(t)\ h(s_{c_j^v}(t))$
  net input and activations of output units (9):
    $net_k(t) = \sum_m w_{km}\ y^m(t-1)\ ,\quad y^k(t) = f_k(net_k(t))$
  derivatives for input gates, forget gates and cells
    variable $dS_{lm}^{jv}(t) := \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}}\ ,\ l \in \{\varphi, in, c\}\ ,\ l \in \{\varphi_j, in_j, c_j^v\}$
    input gates (20): $dS_{in,m}^{jv}(0) = 0,$
    $dS_{in,m}^{jv}(t) = dS_{in,m}^{jv}(t-1)\ y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ f'_{in_j}(net_{in_j}(t))\ y^m(t-1)$
    forget gates (21): $dS_{\varphi m}^{jv}(0) = 0,$
    $dS_{\varphi m}^{jv}(t) = dS_{\varphi m}^{jv}(t-1)\ y^{\varphi_j}(t) + h(s_{c_j^v}(t))\ f'_{\varphi_j}(net_{\varphi_j}(t))\ y^m(t-1)$
    cells (19): $dS_{cm}^{jv}(0) = 0,$
    $dS_{cm}^{jv}(t) = dS_{cm}^{jv}(t-1)\ y^{\varphi_j}(t) + g'(net_{c_j^v}(t))\ y^{in_j}(t)\ y^m(t-1)$
backward pass if error injected
  errors and δs
    injection error (12): $e_k(t) := t^k(t) - y^k(t)$
    output units (14): $\delta_k(t) = f'_k(net_k(t))\ e_k(t)$
    output gates (15):
    $\delta_{out_j}(t) = f'_{out_j}(net_{out_j}(t))\ \left(\sum_{v=1}^{S_j} h(s_{c_j^v}(t)) \sum_k w_{kc_j^v}\ \delta_k(t)\right)$
    input gates, forget gates and cells (17):
    $e_{s_{c_j^v}}(t) = y^{out_j}(t)\quad h'(s_{c_j^v}(t)) \left(\sum_k w_{kc_j^v}\ \delta_k(t)\right)$
  weight updates
    output units and output gates (13):
    $\Delta w_{lm}(t) = \alpha\ \delta_l(t)\ y^m(t-1)\ ,\ l \in \{k, i, out\}$
    input gates (24): $\Delta w_{in,m}(t) = \alpha\ \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}(t)\quad dS_{in,m}^{jv}(t)$
    forget gates (24): $\Delta w_{\varphi m}(t) = \alpha\ \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}(t)\quad dS_{\varphi m}^{jv}(t)$
    cells (23): $\Delta w_{c_j^v m}(t) = \alpha\ e_{s_{c_j^v}}(t)\ dS_{cm}^{jv}(t)$
UNTIL error stopping criterion

References

Bengio, Y., Simard, P., and 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., and McClelland, J. L. (1989). Finite-state automata and simple recurrent networks. *Neural Computation*, 1:372–381.

Cummins, F., Gers, F., and Schmidhuber, J. (1999). Automatic discrimination among languages based on prosody alone. Technical Report IDSIA-03-99, IDSIA, Lugano, CH.

Doya, K. and Shuji, Y. (1989). Adaptive neural oscillator using continuous-time backpropagation learning. *Neural Networks*, 2(5):375–385.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Fahlman, S. E. (1991). The recurrent cascade-correlation learning algorithm. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *NIPS 3*, pages 190–196. San Mateo, CA: Morgan Kaufmann.

Haffner, P. and Waibel, A. (1992). Multi-state time delay networks for continuous speech recognition. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems*, volume 4, pages 135–142. Morgan Kaufmann Publishers, Inc.

Hinton, G. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cogni tive Science Society*, pages 1–12, Amherst 1986. Lawrence Erlbaum, Hillsdale.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München. See www7.informatik.tu-muenchen.de/~hochreit.

Hochreiter, S. and 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., and 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.

Mozer, M. C. (1993). Neural net architectures for temporal sequences processing. In Weigend, A. S. and Gershenfeld, N. A., editors, *Time series prediction: Forecasting the future and understanding the past*, volume 15, pages 243–264. Addison Wesley, Reading, MA.

Mozer, M. C. and Soukup, T. (1991). Connectionist music composition based on melodic and stylistic constraints. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems*, volume 3, pages 789–796. Morgan Kaufmann Publishers, Inc.

O'Reilly, R. C., Braver, T. S., and Cohen, J. D. (1999). A biologically based computational model of working memory. In Miyake, A. and Shah, P., editors, *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*. in press.

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. and Fallside, F. (1987). The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, 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.

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

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

Sutton, R. S. (1988). Learning to predict by methods of temporal differences. *Machine Learning*, 3:9–44.

Tani, J. and Nolfi, S. (1998). Learn to perceive the world as articulated: An approach for hierarchical learning. In *Proceedings of the Fifth International Conference of the Society for Adaptive Behavior*, pages 633–639. The MIT Press.

Tsioutsias, D. I. and Mjolsness, E. (1996). A mulitscale attentional framework for relaxation neural networks. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems*, volume 8, pages 633–639. The MIT Press.

Tsoi, A. C. and Back, A. D. (1994). Locally recurrent globally feedforward networks: A critical review of architectures. *IEEE Transactions on Neural Networks*, 5(2):229–239.

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

Weigend, A. S., Rumelhart, D. E., and Huberman, B. A. (1991). Generalization by weight-elimination with application to forecasting. In *NIPS 3*, pages 875–882.

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

Williams, R. J. and 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. and Zipser, D. (1992). Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Back-propagation: Theory, Architectures and Applications*. Hillsdale, NJ: Erlbaum.