# LSTM Recurrent Networks Learn Simple Context-Free and Context-Sensitive Languages

Felix A. Gers and Jürgen Schmidhuber

***Abstract*—Previous work on learning regular languages from exemplary training sequences showed that long short-term memory (LSTM) outperforms traditional recurrent neural networks (RNNs). Here we demonstrate LSTMs superior performance on context-free language (CFL) benchmarks for RNNs, and show that it works even better than previous hardwired or highly specialized architectures. To the best of our knowledge, LSTM variants are also the first RNNs to learn a simple context-sensitive language (CSL), namely $a^n b^n c^n$.**

***Index Terms*—Context-free languages (CFLs), context-sensitive languages (CSLs), long short-term memory (LSTM), recurrent neural networks (RNNs).**

## I. Introduction

RECURRENT neural networks (RNNs) are remarkably general sequence processing devices [22]. In principle they are applicable to tasks beyond the reach of hidden Markov models (HMMs) or discrete symbolic grammar learning algorithms (SGLAs) [9], [18]. For instance, unlike RNNs, both HMMs and SGLAs are limited to discrete state spaces. Unlike RNNs, HMMs typically ignore information conveyed by the size of temporal delays between relevant events (e.g., in rhythmic patterns). Although the HMM designer could in principle deal with a finite set of time delays between possible observations by devoting a separate internal state for each possible delay, in general this would be cumbersome and inefficient. Unlike RNNs, SGLAs cannot deal well with noisy input sequences[12]. RNNs, however, perform gradient descent in a very general, continuous space of potentially noise-resistant algorithms, using distributed internal memories for mapping real-valued input sequences to real-valued output sequences (noise tends to make learning harder though—compare, e.g., [10] and [11]).

Until recently, however, standard RNNs [13] have been plagued by a major practical problem: the gradient of the total output error with respect to previous inputs quickly vanishes as the time lags between relevant inputs and errors increase [6], [7], [32]. Hence standard RNNs fail to learn in the presence of time lags exceeding as few as five to ten discrete-time steps between relevant input events and target signals [6], [20].

The recent long short-term memory (LSTM) method [7], however, is not affected by this problem. LSTM can learn to bridge minimal time lags in excess of 1000 discrete-time steps [7] by enforcing *constant* error flow through constant error carousels (CECs) within special units, without loss of short-time lag capabilities. Multiplicative gate units learn to open and close access to the constant error flow. Moreover, LSTMs learning algorithm is more efficient than previous RNN algorithms such as real-time recurrent learning (RTRL) [15], [30] and backpropagation through time (BPTT) [27], [29]: it is local in space and time, with computational complexity $O(1)$ per time step and weight.

Recent research on LSTM has concentrated on improving the structure of the adaptive gates surrounding the CECs. Here we will focus on our most recent LSTM variant [4], which provides the gates with direct access to the CEC states, can learn to selectively reset its own memory contents, and can produce stable results in presence of never-ending input streams.

Previous work showed that LSTM outperforms traditional RNN algorithms on numerous tasks involving real-valued or discrete inputs and targets [5], [7], including tasks that require to learn the rules of regular languages (RLs) describable by deterministic finite-state automata (DFA) [1], [2], [8], [21], [31]. Until now, however, it has remained unclear whether LSTMs superiority carries over to tasks involving context-free languages (CFLs), such as those discussed in the RNN literature [16], [17], [23]–[25], [28]. Their recognition requires the functional equivalent of a stack. It is conceivable that LSTM has just the right bias for RLs but might fail on CFLs.

Here we will focus on the most common CFLs benchmarks found in the RNN literature: $a^n b^n$ and $a^n b^m B^m A^n$. We study questions such as:

- Can LSTM learn the functional equivalent of a pushdown automaton?
- Given training sequences up to size $n$, can it generalize to $n+1, n+2, \ldots$?
- How stable are the solutions?
- Does LSTM outperform previous approaches?

Finally, we will apply LSTM to a contex-sensitive language (CSL). The CSLs include the CFLs, which include the RLs. We will focus on the classic example $a^n b^n c^n$, which is a CSL but not a CFL (Section III). In general, CSL recognition requires a linear-bounded automaton, a special Turing machine whose tape length is at most linear in the input size. The $\{a^n b^n c^n\}$ language is one of the simplest CSLs; it can be generated by a tree-adjoined grammar and recognized using a so-called embedded push-down automaton [26] or a finite-state automaton with access to two counters that can be incremented or decremented. To our knowledge no RNN has been able to learn a CSL.

Manuscript received January 8, 2000; revised November 27, 2000 and February 7, 2001. This work was supported by SNF under Grant 2100-49'144.96 "Long Short-Term Memory."

The authors are with the IDSIA, 6928 Manno, Switzerland (e-mail: felix@idsia.ch; juergen@idsia.ch).

Publisher Item Identifier S 1045-9227(01)03569-X.

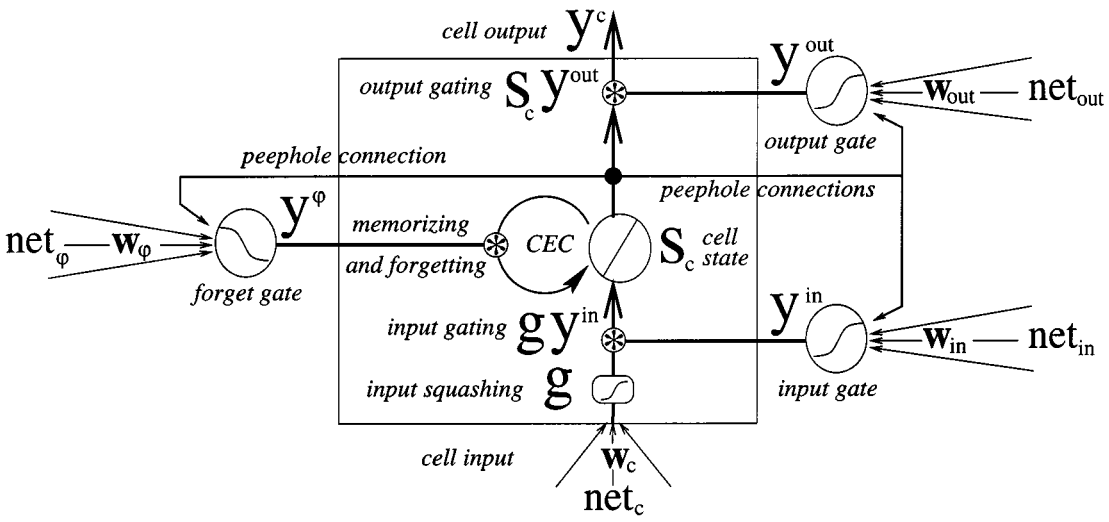


Fig. 1. LSTM memory block with one cell.

## II. LSTM

We are using LSTM with forget gates and the recently introduced peephole connections [4]. Forget gates were shown to be essential for problems involving continual or very long input strings [5]. Peephole connections allow the gates to access the CEC of the same memory block. This proved to be essential for numerous symbolic and real-valued counting tasks [4]. The LSTM variant with peepholes and forget gates used in this paper is clearly superior to traditional LSTM, and is now our method of choice.

The basic unit of an LSTM network is the *memory block* containing one or more *memory cells* and three adaptive, multiplicative gating units shared by all cells in the block (Fig. 1). Each memory cell has at its core a recurrently self-connected linear unit called the CEC whose activation is called the cell *state*. The CECs enforce *constant* error flow and overcome a fundamental problem plaguing previous RNNs: they prevent error signals from decaying quickly as they "back in time." The adaptive gates control input and output to the cells (*input and output gate*) and learn to reset the cell's state once its contents are out of date (*forget gate*). The gates are units whose outputs multiplicatively influence connections to or from the linear unit holding the cell state. Peephole connections connect the CEC to the gates. 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, while gates etc. learn the nonlinear aspects of sequence processing. This makes LSTMs updates efficient without significantly affecting learning power: The CECs permit LSTM to bridge huge time lags (1000 discrete-time steps and more) between relevant events, while traditional RNNs already fail to learn in the presence of ten step time lags, despite requiring more complex update algorithms. Moreover, LSTM is local in space and time [19], and it is more efficient than other RNN algorithms such as RTRL [15], [30]. LSTMs computational complexity for a memory block $j$ per time step and weight is $O(S_j)$, where $S_j$ is the number of cells in the block. $S_j$ is typically a small constant—in this paper it is in fact always equal to one, so that the computational complexity is $O(1)$. BPTT [27], [29] has the same computational complexity (whereas RTRL is much worse) but is not local in time. It needs to store activation values observed during sequence processing in a stack with potential unlimited size. Essentially, LSTM has the complexity of BPTT$(h)$ [30], which truncates errors after $h$ steps, and which works as well as BPTT, according to our experience—if we did not have LSTM we would in fact use BPTT $(h)$.

### A. Forward Pass

The cell output, $y^c$, is calculated based on the current cell state $s_c$ and four sources of input: $\mathrm{net}_c$ is input to the cell itself while $\mathrm{net}_{\mathrm{in}}$, $\mathrm{net}_\varphi$ and $\mathrm{net}_{\mathrm{out}}$ are inputs to the input gates, forget gates, and output gates. We consider discrete-time steps $t = 0, 1, 2, \ldots$. A single step involves the update of all units (forward pass) and the computation of error signals for all weights (backward pass). Throughout this paper, $j$ indexes memory blocks; $v$ indexes memory cells in block $j$ (with $S_j$ cells), 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, and index $m_j$ ranges over all source units and also the CECs of the $j$th memory block. For the gates, $f_l$, $l \in \{\mathrm{in}, \mathrm{out}, \varphi\}$ is a logistic sigmoid with range [0, 1].

For each discrete-time step we use a two-phase update scheme that computes (in this order):

1)
   a) input gate activation $y^{\mathrm{in}}$;
   b) forget gate activation $y^{\varphi}$;
   c) cell input and cell state $s_c$.
2) output gate activation $y^{\mathrm{out}}$ and cell output $y^c$.

**Step 1a,1b**) The input gate activation $y^{\mathrm{in}}$ and the forget gate activation $y^{\varphi}$ are computed as follows:

$$\mathrm{net}_{\mathrm{in}_j}(t) = \sum_m w_{\mathrm{in}_j m} y^m(t-1) + \sum_{v=1}^{S_j} w_{\mathrm{in}_j c_j^v} s_{c_j^v}(t-1)$$
$$y^{\mathrm{in}_j}(t) = f_{\mathrm{in}_j}\left(\mathrm{net}_{\mathrm{in}_j}(t)\right) \quad (1)$$

$$\text{net}_{\varphi_j}(t) = \sum_m w_{\varphi_j m} y^m(t-1) + \sum_{v=1}^{S_j} w_{\varphi_j c_j^v} s_{c_j^v}(t-1)$$
$$y^{\varphi_j}(t) = f_{\varphi_j}\left(\text{net}_{\varphi_j}(t)\right). \quad (2)$$

The peephole connections for the input gate and the forget gate are incorporated in (1) and (2) and by including the CECs (containing the cell states) of memory block $j$ as source units.

**Step 1c**) The state of memory cell $s_c(t)$ is calculated by adding the squashed, gated input of the cell to the state at the previous time step $s_c(t-1)$ $(t > 0)$, which is multiplied (gated) by the forget gate activation $\left(s_{c_j^v}(0) = 0\right)$

$$\text{net}_{c_j^v}(t) = \sum_m w_{c_j^v m} y^m(t-1),$$
$$s_{c_j^v}(t) = y^{\varphi_j}(t) s_{c_j^v}(t-1) + y^{\text{in}_j}(t) g\left(\text{net}_{c_j^v}(t)\right). \quad (3)$$

**Step 2**) The output gate activation $y^{\text{out}}$ is computed as follows:

$$\text{net}_{\text{out}_j}(t) = \sum_m w_{\text{out}_j m} y^m(t-1) + \sum_{v=1}^{S_j} w_{\text{out}_j c_j^v} s_{c_j^v}(t)$$
$$y^{\text{out}_j}(t) = f_{\text{out}_j}\left(\text{net}_{\text{out}_j}(t)\right). \quad (4)$$

Equation (4) includes the peephole connections for the output gate from the CECs of memory block $j$ with the cell states $s_c(t)$, actualized in step 1. The cell output $y^c$ is computed as follows:

$$y^{c_j^v}(t) = y^{\text{out}_j}(t) s_{c_j^v}(t). \quad (5)$$

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

$$\text{net}_k(t) = \sum_m w_{km} y^m(t-1), \quad y^k(t) = f_k(\text{net}_k(t)) \quad (6)$$

where $m$ ranges over all units feeding the output units and $f_k$ is the output squashing function.

### B. Gradient-Based Backward Pass

Essentially, LSTMs backward pass is an efficient fusion of slightly modified, truncated BPTT, and a customized version of RTRL (for details see [5] and [7]). We are using iterative gradient descent, minimizing an objective function $E$, here the usual mean squared error over all time steps and all sequences. Appendix A lists pseudocode for the entire algorithm; see [5] for a full derivation. Unlike BPTT and RTRL, the LSTM learning algorithm is local in space and time.

## III. Experiments

The network sequentially observes exemplary symbol strings of a given language, presented one input symbol at a time. Following the traditional approach in the RNN literature we formulate the task as a prediction task. At any given time step the target is to predict the possible next symbols, including the "end of string" symbol $T$. When more than one symbol can occur in the next step *all* possible symbols have to be predicted, and none of the others.

The network sequentially observes exemplary symbol strings of a given language, presented one input symbol at a time, also referred to as input sequences. Every input sequence begins with the start symbol $S$. The empty string, consisting of $ST$ only, is considered part of each language. A string is accepted when all predictions have been correct. Otherwise it is rejected.

This prediction task is equivalent to a classification task with two classes "accept" and "reject," because the system will make prediction errors for all strings outside the language. A system has learned a given language up to string size $n$ once it is able to correctly predict all strings with size $\leq n$.

Symbols are encoded locally by $d$-dimensional binary vectors with only one nonzero component, where $d$ equals the number of language symbols plus one for either the start symbol in the input or the "end of string" symbol in the output ($d$ input units, $d$ output units). $+1$ signifies that a symbol is set and $-1$ that it is not; the decision boundary for the network output is 0.0.

**CFL** $a^n b^n$ [17], [24], [25], [28]). Here the strings in the input sequences are of the form $a^n b^n$; input and output vectors are three-dimensional. Prior to the first occurrence of $b$ either $a$ or $b$, or $a$ or $T$ at sequence beginnings, are possible in the next step. Thus, e.g., for $n = 5$:

$$\text{Input: } S\ a\ a\ a\ a\ a\ b\ b\ b\ b\ b$$
$$\text{Target: } \frac{a}{T}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ b\ b\ b\ b\ b\ T.$$

**CFL** $a^n b^m B^m A^n$ [16]. The second half of a string from this palindrome or mirror language is completely predictable from the first half. The task involves an intermediate time lag of length $2m$. Input and output vectors are five-dimensional. Prior to the first occurrence of $B$ two symbols are possible in the next step. Thus, e.g., for $n = 4, m = 3$:

$$\text{Input: } S\ a\ a\ a\ a\ b\ b\ b\ B\ B\ B\ A\ A\ A\ A$$
$$\text{Target: } \frac{a}{T}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ \frac{b}{B}\ \frac{b}{B}\ \frac{b}{B}\ B\ B\ A\ A\ A\ A\ T.$$

**CSL** $a^n b^n c^n$. Input and output vectors are four-dimensional. Prior to the first occurrence of $b$ two symbols are possible in the next step. Thus, e.g., for $n = 5$:

$$\text{Input: } S\ a\ a\ a\ a\ a\ b\ b\ b\ b\ b\ c\ c\ c\ c\ c$$
$$\text{Target: } \frac{a}{T}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ \frac{a}{b}\ b\ b\ b\ b\ b\ c\ c\ c\ c\ c\ T.$$

### A. Training and Testing

Learning and testing alternate: after each epoch ($= 1000$ training sequences) we freeze the weights and run a test. Even when all strings are processed correctly during training, it is necessary to test again with frozen weights once all weight changes have been executed. Apart from ensuring the learning of the training set the test also determines generalization performance, which we did not optimize by using, say, a validation set.

Training and test sets incorporate all legal strings up to a given length: $2n$ for $a^n b^n$, $3n$ for $a^n b^n c^n$ and $2(n+m)$ for $a^n b^m B^m A^n$. Training strings are presented in random order. Only exemplars from the class "accept" are presented. Training is stopped once all training sequences have been accepted, or after at most $10^7$ training sequences. The *generalization set* is the largest accepted test set.

Weight changes are made after each sequence. We apply the momentum algorithm [14] with learning rate $\alpha$ is $10^{-5}$ and momentum parameter 0.99. All results are averages over ten independently trained networks with different weight initializations (these ten initializations are identical for each experiment).

**CFL** $a^n b^n$. We study training sets with $n \in \{1, \ldots, N\}$. We test all sets with $n \in \{1, \ldots, M\}$ and $M \in \{N, \ldots, 1000\}$ (sequences of length $\leq 2000$).

**CFL** $a^n b^m B^m A^n$. We use two training sets: 1) The same set as used by [16]: $n \in \{1, \ldots, 11\}$, $m \in \{1, \ldots, 11\}$ with $n + m \leq 12$ (sequences of length $\leq 24$). 2) The set given by $n \in \{1, \ldots, 11\}$, $m \in \{1, \ldots, 11\}$ (sequences of length $\leq 44$). We test all sets with $n \in \{1, \ldots, M\}$, $m \in \{1, \ldots, M\}$ and $M \in \{11, \ldots, 50\}$ (sequences of length $\leq 200$).

**CSL** $a^n b^n c^n$. We study two kinds of training sets: 1) with $n \in \{1, \ldots, N\}$ and 2) with $n \in \{N-1, N\}$. Case 2) asks for a major generalization step that seems almost impossible at first glance: Given very similar training sequences whose sizes differ by at most two, learn to process sequences of arbitrary size! We test all sets with $n \in \{L, \ldots, M\}$, $L \in \{1, \ldots, N-1\}$ and $M \in \{N, \ldots, 500\}$ (sequences of length $\leq 1500$).

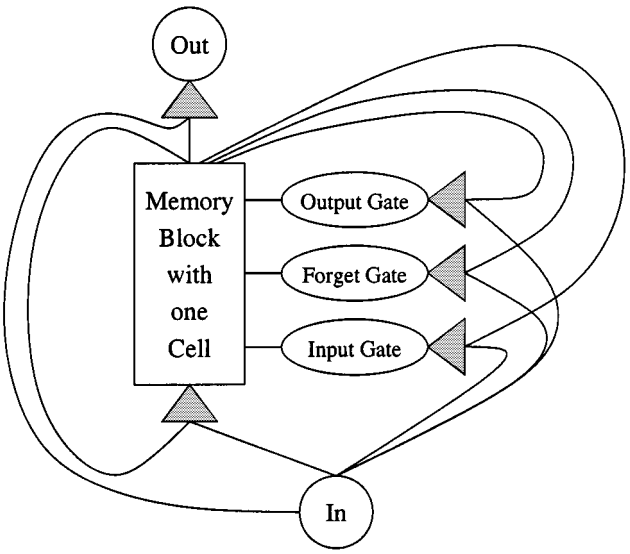


Fig. 2. Three-layer LSTM topology with a single input and output. Recurrence is limited to the hidden layer, consisting here of a single LSTM memory block with a single cell. All 10 "unit-to-unit" connections are shown (but bias and peephole connections are not).

### *B. Network Topology and Experimental Parameters*

The input units are fully connected to a hidden layer consisting of memory blocks with one cell each. The cell outputs are fully connected to the cell inputs, to all gates, and to the output units, which also have direct "shortcut" connections from the input units (Fig. 2). For each task we selected the topology with minimal number of memory blocks that solved the task without extensive parameter optimization. Larger topologies never led to disadvantages except for an increase in computational complexity.

All gates, the cell itself and the output unit are biased. The bias weights to input gate, forget gate and output gate are initialized with $-1.0$, $+2.0$ and $-2.0$, respectively. Although not critical, these values have been found empirically to work well; we use them for all our experiments. With this bias initialization, 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. The forget gates start off closed, so that the cells initially remember everything. We also tried different bias configurations; the results were qualitatively the same, which supports our claim that precise initialization is not critical—see also [5] and [7] for additional evidence in this vein. All other weights are initialized randomly in the range $[-0.1, 0.1]$. The cell's input squashing function $g$ is the identity function. The squashing function of the output units is a sigmoid function with the range $[-2, 2]$.

**CFL** $a^n b^n$. We use one memory block (with one cell). With peephole connections there are 38 adjustable weights (three peephole, 28 unit-to-unit, and seven bias connections).

**CFL** $a^n b^m B^m A^n$. We use two blocks with one cell each, resulting in 110 adjustable weights (six peephole, 91 unit-to-unit, and 13 bias connections).

**CSL** $a^n b^n c^n$. We use the same topology as for the $a^n b^m B^m A^n$ language, but with four input and output units instead of five, resulting in 90 adjustable weights (six peephole, 72 unit-to-unit and 12 bias connections).

### *C. Previous Results*

**CFL** $a^n b^n$. Published results on the $a^n b^n$ language are summarized in Table I.[1,2] RNNs trained with plain BPTT tend to learn to just reproduce the input [17], [25], [28]). Sun *et al.* [24] used a highly specialized architecture, the "neural pushdown automaton," which also did not generalize well [3], [24].

**CFL** $a^n b^m B^m A^n$. Rodriguez and Wiles [16] used BPTT-RNNs with five hidden nodes. After training with $51 \cdot 10^3$ strings with $n + m \leq 12$ (sequences of length $\leq 24$), most networks generalized on longer off-training set strings. The best network generalized to sequences up to length 36 $(n = 9, m = 9)$. But none of them learned the complete training set.

**CSL** $a^n b^n c^n$. To our knowledge no previous RNN ever learned a CSL.

### *D. LSTM Results*

CFL $a^n b^n$. 100% solved for all training sets (Table II). Small training sets $(n \in \{1, \ldots, 10\})$ were already sufficient for perfect generalization up to the tested maximum: $n \in \{1, \ldots, 1000\}$. Note that long sequences of this kind require very stable, finely tuned control of the network's internal counters [2]. This performance is much better than with previous approaches, where the largest set was learned by the specially designed neural push-down automaton [3],

[1]Sun's training set was augmented stepwise by sequences misclassified during testing, and in the final accepted set $n$ was in $\{1, \ldots, 20\}$ except for 20 random sequences up to length $n = 160$ (the exact generalization performance was unclear).

[2]Applying brute force search to the weights of the best network of [16] further improves performance to acceptance up to $n = 28$.

TABLE I
PREVIOUS RESULTS FOR THE CFL $a^n b^n$, SHOWING (FROM LEFT TO RIGHT) THE NUMBER OF HIDDEN UNITS OR STATE UNITS, THE VALUES OF $n$ USED DURING TRAINING, THE NUMBER OF TRAINING SEQUENCES, THE NUMBER OF FOUND SOLUTIONS/TRIALS AND THE LARGEST ACCEPTED TEST SET

| Reference | Hidden Units | Train. Set [$n$] | Train. Str. [$10^3$] | Sol./ Tri. | Best Test [$n$] |
|---|---|---|---|---|---|
| (Sun, Giles, Chen, & Lee, 1993)[1] | 5 | 1,..,160 | 13.5 | 1/1 | 1,..,160 |
| (Wiles & Elman, 1995") | 2 | 1,..,11 | 2000 | 4/20 | 1,..,18 |
| (Tonkes & Wiles, 1997) | 2 | 1,..,10 | 10 | 13/100 | 1,..,12 |
| (Rodriguez, Wiles, & Elman, 1999)[2] | 2 | 1,..,11 | 267 | 8/50 | 1,..,16 |

TABLE II
RESULTS FOR THE $a^n b^n$ LANGUAGE, SHOWING (FROM LEFT TO RIGHT) THE VALUES FOR $n$ USED DURING TRAINING, THE AVERAGE NUMBER OF TRAINING SEQUENCES UNTIL BEST GENERALIZATION WAS ACHIEVED, THE PERCENTAGE OF CORRECT SOLUTIONS AND THE BEST GENERALIZATION (AVERAGE OVER ALL NETWORKS GIVEN IN PARENTHESIS)

| Train. Set [$n$] | Train. Str. [$10^3$] | % Sol. | Generalization Set [$n$] |
|---|---|---|---|
| 1,..,10 | 22 (19) | 100 | 1,..,1000 (1,..,118) |
| 1,..,20 | 18 (19) | 100 | 1,..,587 (1,..,148) |
| 1,..,30 | 16 (19) | 100 | 1,..,1000 (1,..,408) |
| 1,..,40 | 25 (28) | 100 | 1,..,1000 (1,..,628) |
| 1,..,50 | 42 (40) | 100 | 1,..,767 (1,..,430) |

[24]: $n \in \{1, \ldots, 160\}$. The latter, however, required training sequences of the same length as the test sequences. From the training set with $n \in \{1, \ldots, 10\}$ LSTM generalized to $n \in \{1, \ldots, 1000\}$, whereas the best previous result (see Table I) generalized only to $n \in \{1, \ldots, 18\}$ (even with a slightly larger training set: $n \in \{1, \ldots, 11\}$). In contrast to [25], we did not observe our networks forgetting solutions as training progresses. So unlike all previous approaches, LSTM reliably finds solutions that generalize well.

The fluctuations in generalization performance for different training sets in Table II may be due to the fact that we did not optimize generalization performance by using a validation set. Instead we simply stopped each epoch ($= 1000$ sequences) once the training set was learned.

**CFL** $a^n b^m B^m A^n$. Training set a): 100% solved; after $29 \cdot 10^3$ training sequences the best network of ten generalized to at least $n, m \in \{1, \ldots, 22\}$ (all strings up to a length of eight symbols processed correctly); the average generalization set was the one with $n, m \in \{1, \ldots, 16\}$ (all strings up to a length of 64 symbols processed correctly), learned after $25 \cdot 10^3$ training sequences on average.

Training set b): 100% solved; after $26 \cdot 10^3$ training sequences the best network generalized to at least $n, m \in \{1, \ldots, 23\}$ (all strings until a length of 92 symbols processed correctly). The average generalization set was the one with $n, m \in \{1, \ldots, 17\}$ (all strings until a length of 68 symbols processed correctly), learned after $82 \cdot 10^3$ training sequences on average. Unlike the previous approach of [16], LSTM easily learns the complete training set and reliably finds solutions that generalize well.

**CSL** $a^n b^n c^n$. LSTM learns four of the five training sets in ten out of ten trials (only nine out of ten for the training set with $n \in \{1, \ldots, 40\}$) and generalizes well (Table III). Small training sets ($n \in \{1, \ldots, 40\}$) were already sufficient for perfect generalization up to the tested maximum: $n \in \{1, \ldots, 500\}$, that is,

TABLE III
RESULTS FOR THE $a^n b^n c^n$ LANGUAGE, SHOWING (FROM LEFT TO RIGHT) THE VALUES FOR $n$ USED DURING TRAINING, THE AVERAGE NUMBER OF TRAINING SEQUENCES UNTIL BEST GENERALIZATION WAS ACHIEVED, THE PERCENTAGE OF CORRECT SOLUTIONS AND THE BEST GENERALIZATION (AVERAGE OVER ALL NETWORKS IN PARENTHESIS)

| Train. Set [$n$] | Train. Str. [$10^3$] | % Sol. | Generalization Set [$n$] |
|---|---|---|---|
| 1,..,10 | 54 (62) | 100 | 1,..,52 (1,..,28) |
| 1,..,20 | 28 (43) | 100 | 1,..,160 (1,..,66) |
| 1,..,30 | 37 (43) | 100 | 1,..,228 (1,..,91) |
| 1,..,40 | 51 (48) | 90 | 1,..,500 (1,..,120) |
| 1,..,50 | 60 (94) | 100 | 1,..,500 (1,..,409) |
| 10, 11 | 24 (78) | 100 | 9,..,12 (10,..,11) |
| 20, 21 | 829 (626) | 40 | 10,..,27 (17,..,23) |
| 30, 31 | 42 (855) | 30 | 29,..,34 (29,..,32) |
| 40, 41 | 854 (1597) | 40 | 20,..,57 (35,..,45) |
| 50, 51 | 32 (621) | 60 | 43,..,57 (47,..,55) |

sequences of length up to 1500. Even in absence of any short training sequences ($n \in \{N-1, N\}$) LSTM learned well (see bottom half of Table III).

We also modified the training procedure, by presenting each exemplary string without providing all possible next symbols as targets, but only the symbol that actually occurs in the current exemplar. This led to slightly longer training durations, but did not significantly change the results.

### *E. Analysis*

How do the solutions discovered by LSTM work?

**CFL** $a^n b^n$. Fig. 3 shows a test run with a network solution for $n = 5$.

The cell state $s_c$ increases while $a$ symbols are fed into the network, then decreases (with the same step size) while $b$ symbols are fed in. At sequence beginnings (when the first $a$ symbols are observed), however, the step size is smaller due to the closed input gate, which is triggered by $s_c$ itself. This results in "overshooting" the initial value of $s_c$ at the end of a sequence, which in turn triggers the opening of the output gate, which in turn leads to the prediction of the sequence termination.

CFL $a^n b^m B^m A^n$. The behavior of a typical network solution is shown in Fig. 4.

The network learned to establish and control two counters. The two symbol pairs ($a$, $A$) and ($b$, $B$) are treated separately by two different cells, $c_2$ and $c_1$, respectively. Cell $c_2$ tracks the difference between the number of observed $a$ and $A$ symbols. It opens only at the end of a string, where it predicts the final $T$. Cell $c_1$ treats the embedded $b^m B^m$ substring in a similar way.

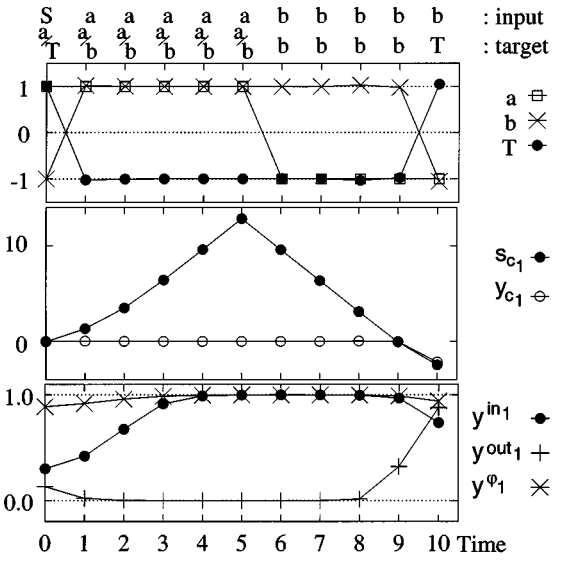


Fig. 3. CFL $a^n b^n$ ($n = 5$): test run with network solutions. Top: network output $y_k$. Middle: Cell state $s_c$ and cell output $y_c$. Bottom: activations of the gates (input gate $y_{\text{in}}$, forget gate $y_\varphi$, and output gate $y_{\text{out}}$).

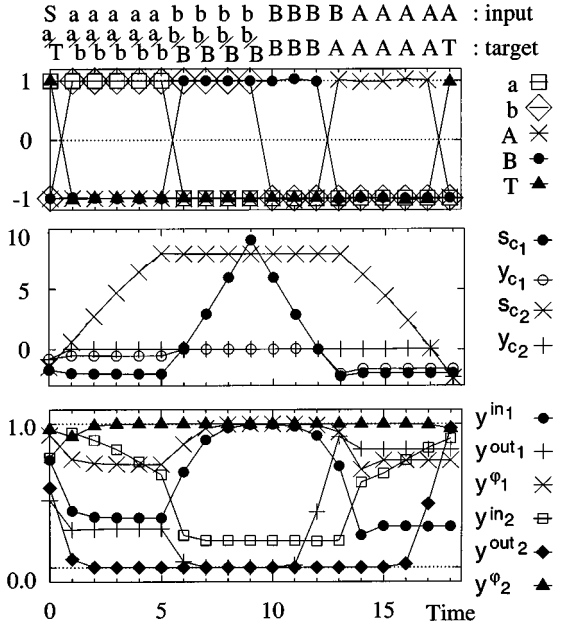


Fig. 4. CFL $a^n b^m B^m A^n$ ($n = 5, m = 4$): test run with network solution. Top: network output $y_k$. Middle: Cell state $s_c$ and cell output $y_c$. Bottom: activations of the gates (input gate $y_{\text{in}}$, forget gate $y_\varphi$, and output gate $y_{\text{out}}$).

While values are stored and manipulated within a cell, the output gate remains closed. This prevents the cell from disturbing the rest of the network and also protects its CEC against incoming errors.

**CFL** $a^n b^n c^n$. The network solutions use a combination of two counters, instantiated separately in the two memory blocks (Fig. 5).

Here the second cell counts up, given an $a$ input symbol. It counts down, given a $b$. A $c$ in the input causes the input gate to close and the forget gate to reset the cell state $s_c$. The second memory block does the same for $b$, $c$, and $a$, respectively. The opening of output gate of the first block indicates the end of a string (and the prediction of the last $T$), triggered via its peephole connection.

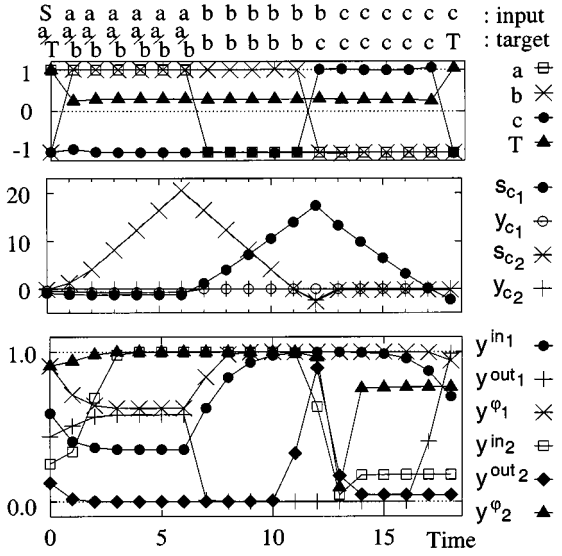


Fig. 5. CSL $a^n b^n c^n$ ($n = 5$): test run with network solution (the system scales up to sequences of length 1000 and more). Top: network output $y_k$. Middle: cell state $s_c$ and cell output $y_c$. Bottom: activations of the gates (input gate $y_{\text{in}}$, forget gate $y_\varphi$, and output gate $y_{\text{out}}$).

Why does the network not generalize for short strings when using only two training strings as for the $a^n b^n c^n$ language (see Table III)? The gate activations in Fig. 5 show that activations slightly drift even when the input stays constant. Solutions take this state drift into account, and will not work without it or with too much of it, as in the case when the sequences are much shorter or longer than the few observed training examples. This imposes a limit on generalization in *both* directions (toward longer and shorter strings). We found solutions with less drift to generalize better.

*Further improvements:* Even better results can be obtained through increased training time and stepwise reduction of the learning rate, as done in [17]. The distribution of lengths of sequences in the training set also affects learning speed and generalization. A set containing more long sequences improves generalization for longer sequences. Omitting the sequence with $n = 1$ (and $m = 1$), typically the last one to be learned, has the same effect. Training sets with many short and many long sequences are learned more quickly than uniformly distributed ones.

*Related tasks:* The $(ba^k)^n$ regular language is related to $a^n b^n$ in the sense that it requires to learn a counter, but the counter never needs counting down. This task is equivalent to the "continual spike timing" task [4] learned by LSTM for $k = 50$ with $n \geq 1000$. A hand-made, hardwired solution (no learning) of a second-order RNN worked for values of $k$ until 120 [23].

For all three tasks peephole connections are mandatory. The output gates remain closed for substantial time periods during each input sequence presentation (compare Figs. 3–5); the end of such a period is always triggered via peephole connections.

## IV. Conclusion

We found that LSTM clearly outperforms previous RNNs not only on regular language benchmarks (according to previous research) but also on CFL benchmarks. It learns faster and generalizes better. LSTM also is the first RNN to learn a CSL.

Although CFLs like those studied in this paper may also be learnable by certain discrete SGLAs [9], [12], [24], the latter exhibit much more task-specific bias, and are not designed to solve numerous other sequence processing tasks involving noise, real-valued inputs/internal states, and continuous output trajectories, which LSTM solves easily [5], [7].

Our findings reinforce the perception that LSTM is a very general and promising adaptive sequence processing device, with a wider field of potential applications than alternative RNNs.

## Appendix

### A. Peephole LSTM With Forget Gates in Pseudocode

**init network:**

**reset: CECs:** $s_{c_j^v} = \hat{s}_{c_j^v} = 0$; **partials:** $dS = 0$;

**activations:** $y = \hat{y} = 0$;

**forward pass:**

**input units:** $y =$ current external input;

**roll over:**

**activations:** $\hat{y} = y$; **cell states:** $\hat{s}_{c_j^v} = s_{c_j^v}$;

loop over memory blocks, indexed $j$ {

**Step 1a: input gates (1):**

$$\text{net}_{\text{in}_j} = \sum_m w_{\text{in}_j m}\hat{y}^m + \sum_{v=1}^{S_j} w_{\text{in}_j c_j^v}\hat{s}_{c_j^v}; \quad y^{\text{in}_j} = f_{\text{in}_j}\left(\text{net}_{\text{in}_j}\right);$$

**Step 1b: forget gates (2):**

$$\text{net}_{\varphi_j} = \sum_m w_{\varphi_j m}\hat{y}^m + \sum_{v=1}^{S_j} w_{\varphi_j c_j^v}\hat{s}_{c_j^v}; \quad y^{\varphi_j} = f_{\varphi_j}\left(\text{net}_{\varphi_j}\right);$$

**Step 1c: CECs, i.e., the cell states (3):**

loop over the $S_j$ cells in block $j$, indexed $v$ {

$$\text{net}_{c_j^v} = \sum_m w_{c_j^v m}\hat{y}^m; \quad s_{c_j^v} = y^{\varphi,j}\hat{s}_{c_j^v} + y^{\text{in}_j} g\left(\text{net}_{c_j^v}\right); \quad \}$$

**Step 2:**

**output gate activation: (4):**

$$\text{net}_{\text{out}_j} = \sum_m w_{\text{out}_j m}\hat{y}^m + \sum_{v=1}^{S_j} w_{\text{out}_j c_j^v} s_{c_j^v};$$

$$y^{\text{out}_j} = f_{\text{out}_j}(\text{net}_{\text{out}_j});$$

**cell outputs (5):**

loop over the $S_j$ cells in block $j$, indexed $v$ $\{y^{c_j^v} = y^{\text{out}_j} s_{c_j^v};\}$

} end loop over memory blocks

**output units (6):**

$\text{net}_k = \sum_m w_{km} y^m$; $y^k = f_k(\text{net}_k)$;

**partial derivatives:**

loop over memory blocks, indexed $j$ {

loop over the $S_j$ cells in block $j$, indexed $v$ {

**cells,** $\left(dS_{cm}^{jv} := \partial s_{c_j^v}/\partial w_{c_j^v m}\right)$:

$$dS_{cm}^{jv} = dS_{cm}^{jv} y^{\varphi_j} + g'\left(\text{net}_{c_j^v}\right) y^{\text{in}_j}\hat{y}^m;$$

**input gates,** $\left(dS_{\text{in},m}^{jv} := \partial s_{c_j^v}/\partial w_{\text{in}_j m},\ dS_{\text{in},c_j^{v'}}^{jv} := \partial s_{c_j^v}/\partial w_{\text{in}_j c_j^{v'}}\right)$:

$$dS_{\text{in},m}^{jv} = dS_{\text{in},m}^{jv} y^{\varphi j} + g\left(\text{net}_{c_j v}\right) f'_{\text{in}_j}\left(\text{net}_{\text{in}_j}\right)\hat{y}^m;$$

loop over peephole connections from all cells, indexed $v'$ {

$$dS_{\text{in},c_j^{v'}}^{jv} = dS_{\text{in},c_j^{v'}}^{jv} y^{\varphi_j} + g\left(\text{net}_{c_j^v}\right) f'_{\text{in}_j}\left(\text{net}_{\text{in}_j}\right)\hat{s}_c^{v'}; \quad \}$$

**forget gates,** $\left(dS_{\varphi m}^{jv} := \partial s_{c_j^v}/\partial w_{\varphi_j m},\ dS_{\varphi c_j^{v'}}^{jv} := \partial s_{c_j^v}/\partial w_{\varphi_j c_j^{v'}}\right)$:

$$dS_{\varphi m}^{jv} = dS_{\varphi m}^{jv} y^{\varphi_j} + \hat{s}_{c_j^v} f'_{\varphi_j}\left(\text{net}_{\varphi_j}\right)\hat{y}^m;$$

loop over peephole connections from all cells, indexed $v'$ {

$$dS_{\varphi c_j^{v'}}^{jv} = dS_{\varphi c_j^{v'}}^{jv} y^{\varphi_j} + \hat{s}_{c_j^v} f'_{\varphi_j}\left(\text{net}_{\varphi_j}\right)\hat{s}_c^{v'}; \quad \}$$

} } end loops over cells and memory blocks

**backward pass (if error injected):**

**errors and** $\delta$s:

**injection error:** $e_k = t^k - y^k$;

$\delta$s **of output units:** $\delta_k = f'_k(\text{net}_k) e_k$;

loop ever memory blocks, indexed $j$ {

$\delta$s **of output gates:**

$$\delta_{\text{out}_j} = f'_{\text{out}_j}(\text{net}_{\text{out}_j})\left(\sum_{v=1}^{S_j} s_{c_j^v} \sum_k W_{k c_j^v}\delta_k\right);$$

**internal state error:**

loop over the $S_j$ cells in block $j$, indexed $v$ {

$$e_{s_{c_j^v}} = y^{\text{out}_j}\left(\sum_k w_{k c_j^v}\delta_k\right); \quad \}$$

} end loop over memory blocks

**weight updates:**

**output units:** $\Delta w_{km} = \alpha\delta_k y^m$;

loop over memory blocks, indexed $j$ {

**output gates:**

$$\Delta w_{\text{out},m} = \alpha\delta_{\text{out}}\hat{y}^m; \quad \Delta w_{\text{out},c_j^v} = \alpha\delta_{\text{out}} s_{c_j^v};$$

**input gates:**

$$\Delta w_{\text{in},m} = \alpha\sum_{v=1}^{S_j} e_{s_{c_j^v}} dS_{\text{in},m}^{jv};$$

loop over peephole connections from all cells, indexed $v'$ {

$$\Delta w_{\text{in},c_j^{v'}} = \alpha\sum_{v=1}^{S_j} e_{s_{c_j^v}} dS_{\text{in},c_j^{v'}}^{jv}; \quad \}$$

**forget gates:**

$$\Delta w_{\varphi m} = \alpha\sum_{v=1}^{S_j} e_{s_{c_j^v}} dS_{\varphi m}^{jv};$$

loop over peephole connections from all cells, indexed $v'$ {

$$\Delta w_{\varphi c_j^{v'}} = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}} dS^{jv}_{\varphi c_j^{v'}}; \quad \}$$

**cells:**

loop over the $S_j$ cells in block $j$, indexed $v$ {

$$\Delta w_{c_j^v m} = \alpha e_{s_{c_j^v}} dS^{jv}_{cm}; \quad \};$$

end loop over memory blocks

## REFERENCES


[1] A. D. Blair and J. B. Pollack, "Analysis of dynamical recognizers," *Neural Comput*, vol. 9, no. 5, pp. 1127–1142, 1997.

[2] M. P. Casey, "The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction," *Neural Comput.*, vol. 8, no. 6, pp. 1135–1178, 1996.

[3] S. Das, C. Giles, and G. Sun, "Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory," in *Proc. 14th Annu. Conf. Cognitive Sci. Soc.*, San Mateo, CA, 1992, pp. 791–795.

[4] F. A. Gers and J. Schmidhuber, "Recurrent nets that time and count," in Proc. IJCNN'2000, Int. Joint Conf. Neural Networks, Como, Italy, 2000.

[5] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, 2000.

[6] S. Hochreiter, "Untersuchungen zu Dynamischen Neuronalen Netzen," Diploma thesis, Inst. Informatik, Technische Univ. München, München, Germany, 1991.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[8] Y. Kalinke and H. Lehmann, "Computation in recurrent neural networks: From counters to iterated function systems," in *Proc. 11th Australian Joint Conf. Artificial Intell., Advanced Topics Artificial Intell.*, vol. 1502, G. Antoniou and J. Slaney, Eds., Berlin, Germany, 1998.

[9] L. Lee, "Learning languages: Arature," Center Res. Comput. Technol., Harvard Univ., Cambridge, MA, TR-12–96, 1996.

[10] W. Maass and P. Orponen, "On the effect of analog noise in discrete-time analog computations," *Neural Comput.*, vol. 10, no. 5, pp. 1071–1095, 1998.

[11] W. Maass and E. D. Sontag, "Analog neural nets with Gaussian or other common noise distribution cannot recognize arbitrary regular languages," *Neural Comput.*, vol. 11, no. 3, pp. 771–782, 1999.

[12] M. Osborne and E. Briscoe, "Learning stochastic categorial grammars," in *Proc. Assoc. Comp. Linguistics, Comp. Nat. Lg.Learing (CoNLL97) Workshop*, Madrid, Spain, 1997, pp. 80–87.

[13] B. A. Pearlmutter, "Gradient calculations for dynamic recurrent neural networks: A survey," *IEEE Trans. Neural Networks*, vol. 6, pp. 1212–1228, Sept. 1995.

[14] D. C. Plaut, S. J. Nowlan, and G. E. Hinton, "Experiments on learning backpropagation," Carnegie-Mellon University, Pittsburgh, PA, CMU-CS-86–126, 1986.

[15] A. J. Robinson and F. Fallside, "The Utility Driven Dynamic Error Propagation Network," Cambridge Univ., Eng. Dep., CUED/F-INFENG/TR.1, 1987.

[16] P. Rodriguez and J. Wiles, "Recurrent neural networks can learn to implement symbol-sensitive counting," in *Advances in Neural Information Processing Systems*. Cambridge, MA: MIT Press, 1998, vol. 10, pp. 87–93.

[17] P. Rodriguez, J. Wiles, and J. Elman, "A recurrent neural network that learns to count," *Connection Sci.*, vol. 11, no. 1, pp. 5–40, 1999.

[18] Y. Sakakibara, "Recent advances of grammatical inference," *Theoretical Comput. Sci.*, vol. 185, no. 1, pp. 15–45, 1997.

[19] J. Schmidhuber, "The neural bucket brigade, a local learning algorithm for dynamic feedforward and recurrent networks," *Connection Sci.*, vol. 1, no. 4, pp. 403–412, 1989.

[20] ——, "Learning complex, extended sequences using the principle of history compression," *Neural Comput.*, vol. 4, no. 2, pp. 234–242, 1992.

[21] H. Siegelmann, "Theoretical Foundations of Recurrent Neural Networks," doctoral dissertation, Rutgers Univ., New Brunswick, NJ, 1992.

[22] H. T. Siegelmann and E. D. Sontag, "Turing computability with neural nets," Appl. Math., 1991.

[23] M. Steijvers and P. Grunwald, "A recurrent network that performs a context sensitive prediction task," in Proc. 18th Annu. Conf. Cognitive Sci. Soc.. Hillsdale, NJ, 1996.

[24] G. Z. Sun, C. L. Giles, H. H. Chen, and Y. C. Lee, "The Neural Network Pushdown Automaton: Model, Stack and Learning Simulations," Univ. Maryland, College Park, MD, CS-TR-3118, 1993.

[25] B. Tonkes and J. Wiles, "Learning a context-free task with a recurrent neural network: An analysis of stability," in Proc. 4th Biennial Conf. Australasian Cognitive Sci. Soc., 1997.

[26] K. Vijay-Shanker, "Using descriptions of trees in a tree adjoining grammar," *Comput. Linguistics*, vol. 18, no. 4, pp. 481–517, 1992.

[27] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, vol. 1, pp. 339–356, 1988.

[28] J. Wiles and J. Elman, "Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks," in *Proc. 17th Annu. Conf. Cognitive Sci. Soc.*, Cambridge, MA, 1995, pp. 482–487.

[29] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories [Letter]," *Neural Comput.*, vol. 2, no. 4, pp. 490–501, 1990.

[30] R. J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," in *Backpropagation: Theory, Architectures, and Applications*, Y. Chauvin and D. E. Rumelhart, Eds. Hillsdale, NJ: Lawrence Erlbaum, 1992, pp. 433–486.

[31] Z. Zeng, R. Goodman, and P. Smyth, "Discrete recurrent neural networks for grammatical inference," *IEEE Trans. Neural Networks*, vol. 5, Mar. 1994.

[32] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Networks*, vol. 5, pp. 157–166, Mar. 1994.


**Felix A. Gers** was born in Freiburg, Germany, on November 15, 1970. He received the Diploma degree in physics from the University of Hanover, Germany, in 1995.

He worked on a project on cellular automata at the Advanced Telecommunications Research Institute International (ATR), Kyoto, Japan. He is currently participating in a project on learning algorithms for recurrent networks at IDSIA, Manno, Switzerland. His interests include parallel and distributed processing, artificial intelligence, and learning in biological and artificial systems.

**Jürgen Schmidhuber** was born in 1963. He received the Diploma degree in computer science in 1987, the Ph.D. degree in 1991, and the postdoctoral degree (Habilitation) in 1993, all from the Technische Universität München, München, Germany.

From 1991 to 1992, he was a Postdoctoral Fellow at the University of Colorado, Boulder. He is Co-Director of the Swiss AI Research Institute IDSIA, Manno, which he helped to transform into one of the world's top ten AI labs, according to Business Week Magazine in 1997. He has published more than100 scientific papers and created low-complexity art, a new minimal art form based on algorithmic complexity theory.