Long Short-Term Memory Learns Context Free and Context Sensitive Languages

Felix A. Gers Jürgen Schmidhuber

felix@idsia.ch juergen@idsia.ch

IDSIA, Galleria 2, 6928 Manno, Switzerland, www.idsia.ch

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 LSTM's superior performance on context free language (CFL) benchmarks, 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 context *sensitive* language (CSL), namely $a^n b^n c^n$.

1 Introduction

Recurrent neural networks (RNNs) are remarkably general sequence processing devices (Siegelmann and Sontag, 1991). In principle they are applicable to tasks beyond the reach of Hidden Markov Models (HMMs) or discrete symbolic grammar learning algorithms (SGLAs) (Lee, 1996). 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 (Osborne and Briscoe, 1997). 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.

Until recently, however, standard RNNs (Pearlmutter, 1995) 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. Hence standard RNNs fail to learn in the presence of time lags exceeding as few as 5 - 10 discrete time steps between relevant input events and target signals.

The recent *"Long Short-Term Memory"* (LSTM) method (Hochreiter and Schmidhuber, 1997), however, is not affected by this problem. LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing *constant* error flow through "constant error carousels" (CECs) within special units, without loss of short time lag capabilities. Multiplicative gate units learn to open and close access to the constant error flow. Moreover, LSTM's learning algorithm is more efficient than previous RNN algorithms such as real time recurrent learning (RTRL—Robinson & Fallside, 1987) and back propagation through time (BPTT—Williams & Peng, 1990): 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 (Gers and Schmidhuber, 2000), 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 (Gers et al., 1999; Hochreiter and Schmidhuber, 1997), including tasks that require to learn the rules of regular languages (RLs) describable by deterministic finite state automata (DFA) (Casey, 1996; Siegelmann, 1993; Blair and Pollack, 1997; Kalinke and Lehmann, 1998; Zeng et al., 1994). Until now, however, it has remained unclear whether LSTM's superiority carries over to tasks involving context free languages (CFLs), such as those discussed in the RNN literature (Sun et al., 1993; Wiles and Elman, 1995; Steijvers and Grunwald, 1996; Tonkes and Wiles, 1997; Rodriguez et al., 1999; Rodriguez and Wiles, 1998). 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: $a^n b^n$ 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 context *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 3). In general, CSL recognition requires a linear-bounded automaton, a special Turing machine whose tape length is at most linear in the input size. To our knowledge no RNN has been able to learn a CSL.

2 LSTM

We are using LSTM with forget gates and the recently introduced peephole connections (Gers and Schmidhuber, 2000).

Insert Figure 1 around here.

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 (Figure 1). Each memory cell has at its core a recurrently self-connected linear unit called the "Constant Error Carousel" (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*). 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 LSTM's updates efficient without significantly affecting learning power: LSTM's learning algorithm is local in space and time; its computational complexity per time step and weight is $O(1)$. 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 10 step time lags, despite requiring more complex update algorithms.

2.1 Forward Pass

The cell output, y^c, is calculated based on the current cell state s_c and four sources of input: net_c is input to the cell itself while net_{in}, net_φ and net_{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 , $l \in \{in, out, \varphi\}$ 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^{in},

   (b) forget gate activation y^φ,

   (c) cell input and cell state s_c,

2. output gate activation y^{out} and cell output y^c.

Step 1a,1b. The input gate activation y^{in} and the forget gate activation y^φ are computed as follows:

$$net_{in_j}(t) = \sum_m w_{in_j m}\ y^m(t-1) + \sum_{v=1}^{S_j} w_{in_j c_j^v}\ s_{c_j^v}(t-1)\ , \quad y^{in_j}(t) = f_{in_j}(net_{in_j}(t))\ \ , \tag{1}$$

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m}\ y^m(t-1) + \sum_{v=1}^{S_j} w_{\varphi_j c_j^v}\ s_{c_j^v}(t-1)\ , \quad y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t))\ \ . \tag{2}$$

The peephole connections for the input gate and the forget gate are incorporated in equation 1 and 2 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 to the state at the previous time step $s_c(t-1)$ $(t > 0)$, which is multiplied (gated) by the forget gate activation $(s_{c_j^v}(0) = 0)$:

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m}\ y^m(t-1)\ , \quad s_{c_j^v}(t) = y^{\varphi_j}(t)\ s_{c_j^v}(t-1) + y^{in_j}(t)\ \ g(net_{c_j^v}(t))\ \ . \tag{3}$$

Step 2. The output gate activation y^{out} is computed as follows:

$$net_{out_j}(t) = \sum_m w_{out_j m}\ y^m(t-1) + \sum_{v=1}^{S_j} w_{out_j c_j^v}\ s_{c_j^v}(t)\ , \quad y^{out_j}(t) = f_{out_j}(net_{out_j}(t))\ \ . \tag{4}$$

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:

$$y^{c_j^v}(t) = y^{out_j}(t)\ s_{c_j^v}(t) \quad . \tag{5}$$

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:

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

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

2.2 Gradient-Based Backward Pass

Essentially, LSTM's backward pass (for details see Hochreiter and Schmidhuber (1997) and Gers et. al. (2000)) is an efficient fusion of slightly modified, truncated BPTT and a customized version of RTRL. We are using iterative gradient descent, minimizing an objective function E, here the usual mean squared error function. Appendix A lists pseudo code for the entire algorithm; see (Gers et al., 1999) for a full derivation. Unlike BPTT and RTRL, LSTM's learning algorithm is local in space and time.

3 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 terminal 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.

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 the 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 in d-dimensional vectors, where d is equal to the number of symbols of the given language plus one for either the start symbol in the input or the terminal symbol in the output (d input units, d output units, each standing for one of the symbols). $+1$ signifies that a symbol is set and -1 that it is not set; the decision boundary for the network output is 0.0.

CFL $a^n b^n$ (Sun et al., 1993; Wiles and Elman, 1995; Tonkes and Wiles, 1997; Rodriguez et al., 1999). Here the strings in the input sequences are of the form $a^n b^n$, input and output vectors are 3-dimensional. Before the first occurrence of b either a or b, or a or T at sequence beginings, are possible in the next step. Thus, e.g., for $n=5$:

Input:	S	a	a	a	a	a	b	b	b	b	b
Target:	a/T	a/b	a/b	a/b	a/b	a/b	b	b	b	b	T

CFL $a^n b^m B^m A^n$ (Rodriguez and Wiles, 1998). The second half of a string from this palindrome or mirror language is completely predictable from the first half. This task involves an intermediate time lag of length $2m$. Input and output vectors are 5-dimensional. Before the first occurrence of B two symbols are possible in the next step. Thus, e.g., for $n=4, m=3$:

Input:	S	a	a	a	a	b	b	b	B	B	B	A	A	A	A
Target:	a/T	a/b	a/b	a/b	a/b	b/B	b/B	b/B	B	B	A	A	A	A	T

CSL $a^n b^n c^n$**.** Input and output vectors are 4-dimensional. Before the first occurrence of b two symbols are possible in the next step. Thus, e.g., for $n=5$:

Input:	S	a	a	a	a	a	b	b	b	b	b	c	c	c	c	c
Target:	a/T	a/b	a/b	a/b	a/b	a/b	b	b	b	b	c	c	c	c	c	T

3.1 Training and Testing

Learning and testing alternate: after 1000 training sequences we freeze the weights and run a test. 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 $a^n b^m B^m A^n$. Only positive exemplars 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 (Plaut et al., 1986) with learning rate α is 10^{-5} and momentum parameter 0.99. All results are averages over 10 independently trained networks with different weight initializations (the same for each experiment).

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

CFL $a^n b^m B^m A^n$. We use two training sets: a) The same set as used by Rodriguez and Wiles (1999) : $n \in \{1,..,11\}$, $m \in \{1,..,11\}$ with $n + m \leq 12$ (sequences of length ≤ 24). b) The set given by $n \in \{1,..,11\}$, $m \in \{1,..,11\}$ (sequences of length ≤ 48). We test all sets with $n \in \{1,..,M\}$, $m \in \{1,..,M\}$ and $M \in \{11,..,50\}$ (sequences of length ≤ 200).

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

3.2 Network Topology and Experimental Parameters

The input units are fully connected to a hidden layer consisting of memory blocks with 1 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 (Figure 2).

Insert Figure 2 around here.

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 (these are standard values, which we use for all our experiments; precise initialization is not critical here). 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). Without the 3 peephole connections there are 37 adjustable weights (28 unit-to-unit and 7 bias connections).

CFL $a^n b^m B^m A^n$. We use two blocks (with one cell each), resulting 114 adjustable weights (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 4 input and output units instead of 5, giving 84 adjustable weights (72 unit-to-unit and 12 bias connections).

Table 1: 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 et al., 1993)[1]	5	1,..,160	13.5	1/1	1,..,160
(Wiles and Elman, 1995)	2	1,..,11	2000	4/20	1,..,18
(Tonkes and Wiles, 1997)	2	1,..,10	10	13/100	1,..,12
(Rodriguez et al., 1999)[2]	2	1,..,11	267	8/50	1,..,16

3.3 Previous results

CFL $a^n b^n$. Published results on the $a^n b^n$ language are summarized in Table 1. RNNs trained with plain BPTT tend to learn to just reproduce the input (Wiles and Elman, 1995; Tonkes and Wiles, 1997; Rodriguez et al., 1999). Sun et al. (1993) used a highly specialized architecture, the "neural pushdown automaton", which also did not generalize well (Sun et al., 1993; Das et al., 1992).

CFL $a^n b^m B^m A^n$. Rodriguez and Wiles (1998) used BPTT-RNNs with 5 hidden nodes. After training with $n + m \leq 12$ (sequences of length ≤ 24), the best network generalized to sequences up to length 36 ($n = 9$,$m = 9$). But their networks did not learn the complete training set, and generalization was restricted to few strings with $n, m \in \{1,..,9\}$.

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

3.4 LSTM Results

CFL $a^n b^n$. 100% solved for all training sets (Table 2). Small training sets ($n \in \{1,..,10\}$) were already sufficient for perfect generalization up to the tested maximum: $n \in \{1,..,1000\}$. Note that long sequences of

[1] Sun's training set was augmented stepwise by sequences misclassified during testing, and in the final accepted set n was in $\{1,..,20\}$ 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 Rodriguez et al. (1999) further improves performance to acceptance up to $n=28$.

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)

Table 2: Results for the $a^n b^n$ language, showing (left to right) the values for n used during training, the average number of training sequences until best (average over all networks given in parenthesis) generalization was achieved, the percentage of correct solutions and the best generalization (average over all networks given in parenthesis).

this kind require very stable, finely tuned control of the network's internal counters (Casey, 1996).

This performance is much better than in previous approaches, where the largest set was learned by the specially designed neural push-down automaton (Sun et al., 1993; Das et al., 1992): $n \in \{1,..,160\}$. The latter, however, required training sequences of the same length as the test sequences. From the training set with $n \in \{1,..,10\}$ LSTM generalized to $n \in \{1,..,1000\}$, whereas the best previous result (see Table 1) generalized only to $n \in \{1,..,18\}$ (even with a slightly larger training set: $n \in \{1,..,11\}$).

In contrast to Tonkes and Wiles (1997), we did not observe our networks forgetting solutions as training progresses.

So unlike all previous approaches, LSTM reliably finds solutions that generalize well.

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 10 generalized to at least $n, m \in \{1,..,22\}$ (all strings until a length of 88 symbols processed correctly); the average generalization set was the one with $n, m \in \{1,..,16\}$ (all strings until 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 $n, m \in \{1,..,23\}$ (all strings until a length of 92 symbols processed correctly). The average generalization set was the one with $n, m \in \{1,..,17\}$ (all strings until a length of 68 symbols processed correctly), learned after $82 \cdot 10^3$ training sequences on average.

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)

Table 3: Results for the $a^n b^n c^n$ language, showing (left to right) the values for n used during training, the average number of training sequences until best (average over all networks given in parenthesis) generalization was achieved, the percentage of correct solutions and the best generalization (average over all networks given in parenthesis).

Unlike the previous approach of Rodriguez and Wiles (1998),

LSTM easily learns the complete training set and reliably finds solutions that generalize well.

CSL $a^n b^n c^n$. LSTM learns 4 of the 5 training sets in 10 out of 10 trials (only 9 out of 10 for the training set with $n \in \{1,..,40\}$) and generalizes well (Table 3). Small training sets ($n \in \{1,..,40\}$) were already sufficient for perfect generalization up to the tested maximum: $n \in \{1,..,500\}$, that is, 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 Table3).

3.5 Analysis

How do the solutions discovered by LSTM work?

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

Insert Figure 3 around here.

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 Figure 4.

Insert Figure 4 around here.

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. 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.

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

Insert Figure 5 around here.

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.

Further improvements. Even better results can be obtained through increased training time and stepwise reduction of the learning rate, as done in (Rodriguez et al., 1999). The distribution of lengths of sequences in the training set affects learning speed and generalization, as well. 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, (Gers and Schmidhuber, 2000) 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 (Steijvers and Grunwald, 1996).

4 Conclusion

We found that Long Short-Term Memory (LSTM) clearly outperforms previous RNNs not only on regular language benchmarks (according to previous research) but also on context free language (CFL) benchmarks. It learns faster and generalizes better. LSTM also is the first RNN to learn a context sensitive language.

Although CFLs like those studied in this paper may also be learnable by certain discrete symbolic grammar learning algorithms (SGLAs) (Lee, 1996; Osborne and Briscoe, 1997), the latter exhibit much more task-specific bias, and cannot solve numerous other sequence processing tasks involving noise and real-valued inputs, and internal states, and continuous output trajectories, which LSTM solves easily (Hochreiter and Schmidhuber, 1997; Gers et al., 1999).

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.

Acknowledgment

This work was supported by SNF grant 2100-49'144.96 "Long Short-Term Memory."

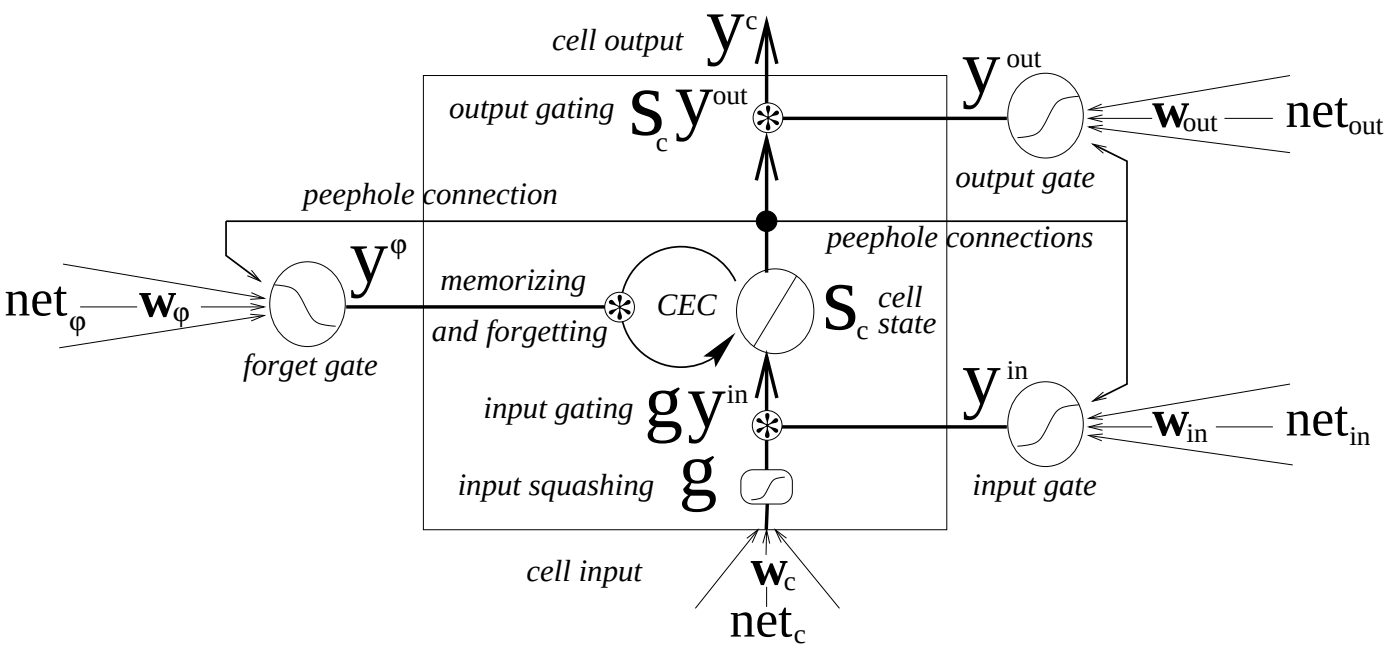

Figure 1: LSTM memory block with one cell.

Gers:Figure 1

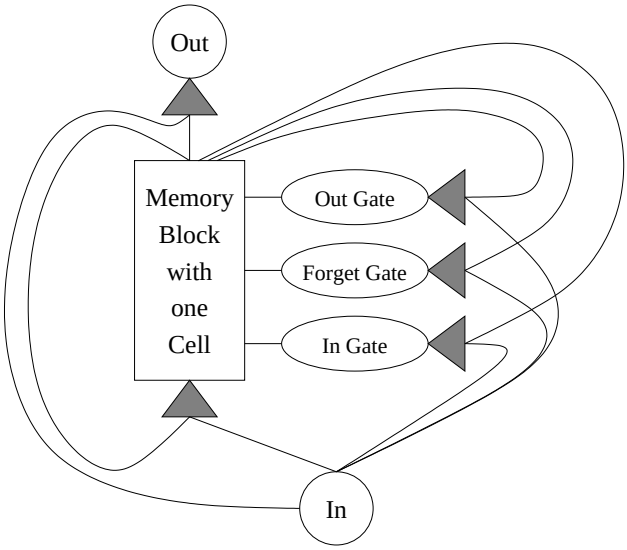

Figure 2: Three-layer LSTM topology with a single input and output. Recurrence is limited to the hidden layer, consisting here of a single LSTM memory block with a single cell. All 10 "unit-to-unit" connections are shown (but bias and peephole connections are not).

Gers: Figure 2

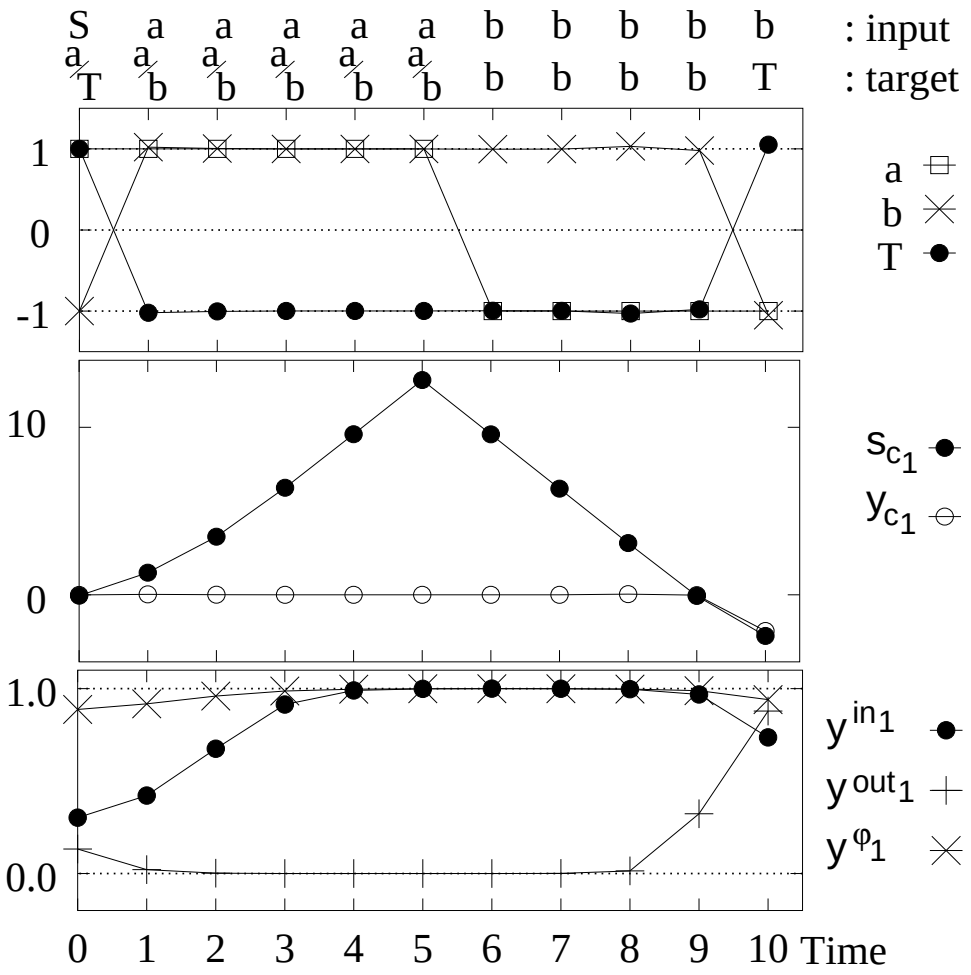

Figure 3: CFL $a^n b^n$ ($n = 5$): Test run with network solutions. Top: Network output y_k. Middle: Cell state s_c and cell output y_c. Bottom: Activations of the gates (input gate y_{in}, forget gate y_φ and output gate y_{out}).

Gers: Figure 3

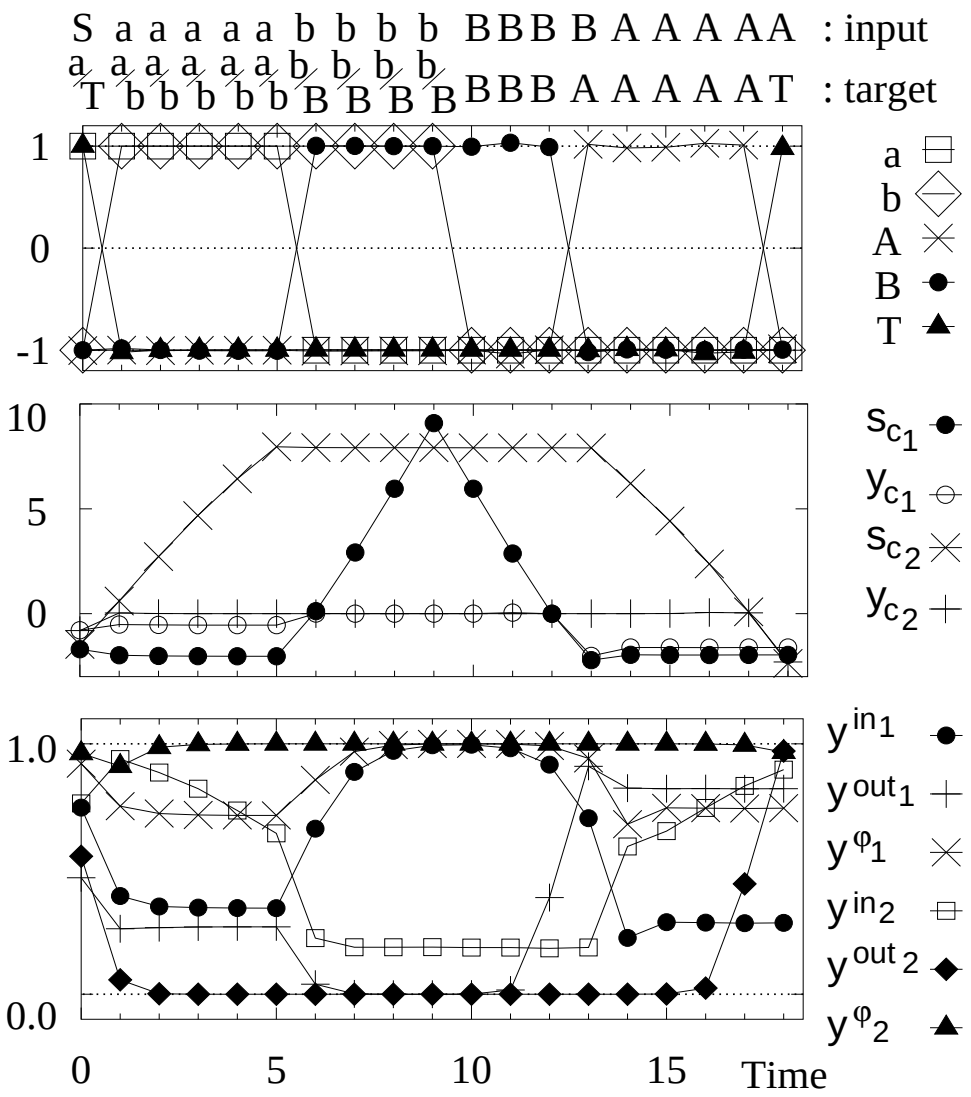

Figure 4: CFL $a^n b^m B^m A^n$ ($n=5, m=4$): Test run with network solution. Top: Network output y_k. Middle: Cell state s_c and cell output y_c. Bottom: Activations of the gates (input gate y_{in}, forget gate y_φ and output gate y_{out}).

Gers: Figure 4

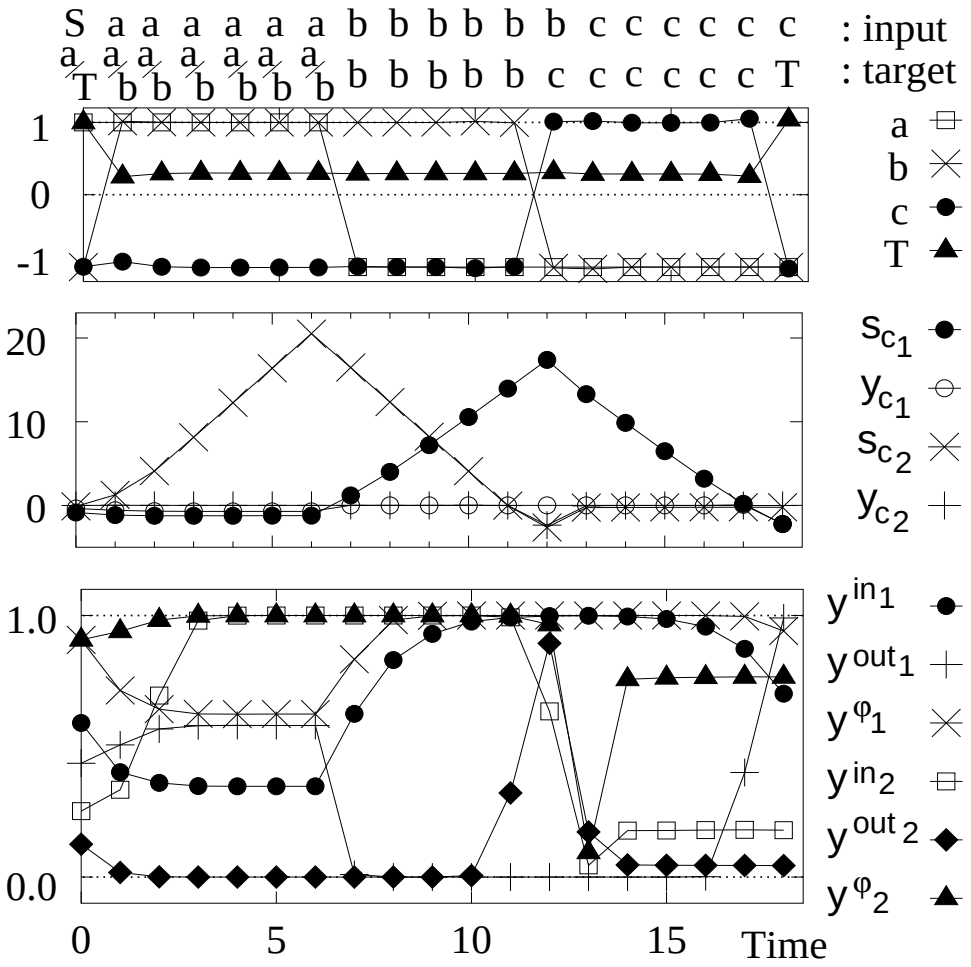

Figure 5: CSL $a^n b^n c^n$ $(n = 5)$: Test run with network solution (the system scales up to sequences of length 1000 and more). Top: Network output y_k. Middle: Cell state s_c and cell output y_c. Bottom: Activations of the gates (input gate y_{in}, forget gate y_φ and output gate y_{out}).

Gers: Figure 5

A Peephole LSTM with Forget Gates in Pseudo-code

REPEAT learning loop

forward pass (net input and activation in hidden layer)

reset all *net* values

Step 1a,1b: input gates and forget gates, $l \in \{\varphi, in\}$: (1), (2):

$net_{l_j}(t) = \sum_m w_{l_j m}\ y^m(t-1) + \sum_{v=1}^{S_j} w_{l_j c_j^v}\ s_{c_j^v}(t-1)\ ,\quad y^{l_j}(t) = f_{l_j}(net_{l_j}(t))$

Step 1c: cell state, $s_{c_j^v}(0) = 0$: (3):

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

Step 2: output gate activation: (4):

$net_{out_j}(t) = \sum_m w_{out_j m}\ y^m(t-1) + \sum_{v=1}^{S_j} w_{out_j c_j^v}\ s_{c_j^v}(t)\ ,\quad y^{out_j}(t) = f_{out_j}(net_{out_j}(t))$

cell output: (5): $y^{c_j^v}(t) = y^{out_j}(t)\ s_{c_j^v}(t)$

output units: (6): $net_k(t) = \sum_m w_{km}\ y^m(t-1)\ ,\quad y^k(t) = f_k(net_k(t))$

derivatives for input gates, forget gates and cells: init all: $dS(0) = 0$

cells : $dS_{cm}^{jv}(t) := \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}}$:

$dS_{cm}^{jv}(t) = dS_{cm}^{jv}(t-1)\ y^{\varphi_j}(t) + g'(net_{c_j^v}(t))\ y^{in_j}(t)\ y^m(t-1)$

input gates : $dS_{in,m}^{jv}(t) := \frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}}$, $dS_{in,s_c^v}^{jv}(t) := \frac{\partial s_{c_j^v}(t)}{\partial w_{in_j s_c^v}}$:

$dS_{in,m}^{jv}(t) = dS_{in,m}^{jv}(t-1)\ y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ f'_{in_j}(net_{in_j}(t))\ y^m(t-1)$

$dS_{in,s_c^v}^{jv}(t) = dS_{in,s_c^v}^{jv}(t-1)\ y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ f'_{in_j}(net_{in_j}(t))\ s_c^v(t-1)$

forget gates : $dS_{\varphi m}^{jv}(t) := \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}}$, $dS_{\varphi s_c^v}^{jv}(t) := \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j s_c^v}}$:

$dS_{\varphi m}^{jv}(t) = dS_{\varphi m}^{jv}(t-1)\ y^{\varphi_j}(t) + s_{c_j^v}(t-1)\ f'_{\varphi_j}(net_{\varphi_j}(t))\ y^m(t-1)$

$dS_{\varphi s_c^v}^{jv}(t) = dS_{\varphi s_c^v}^{jv}(t-1)\ y^{\varphi_j}(t) + s_{c_j^v}(t-1)\ f'_{\varphi_j}(net_{\varphi_j}(t))\ s_c^v(t-1)$

backward pass if error injected

errors (injection error: $e_k(t) := t^k(t) - y^k(t)$ with targets $t^k(t)$) and δs:

output units : $\delta_k(t) = f'_k(net_k(t))\ e_k(t)$

output gates : $\delta_{out_j}(t) = f'_{out_j}(net_{out_j}(t))\ \left(\sum_{v=1}^{S_j}\ s_{c_j^v}(t) \sum_k w_{k c_j^v}\ \delta_k(t)\right)$

input gates, forget gates and cells :

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

weight updates

output units : $\Delta w_{km}(t) = \alpha\ \delta_k(t)\ y^m(t-1)$

output gates :

$\Delta w_{out,m}(t) = \alpha\ \delta_{out}(t)\ y^m(t-1)\ ,\ \Delta w_{out,c_j^v}(t) = \alpha\ \delta_{out}(t)\ s_{c_j^v}(t)$

input gates :

$\Delta w_{in,m}(t) = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}(t)\ dS_{in,m}^{jv}(t)\ ,\ \Delta w_{in,s_c^v}(t) = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}(t)\ dS_{in,s_c^v}^{jv}(t)$

forget gates :

$\Delta w_{\varphi m}(t) = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}(t)\ dS_{\varphi m}^{jv}(t)\ ,\ \Delta w_{\varphi s_c^v}(t) = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}(t)\ dS_{\varphi s_c^v}^{jv}(t)$

cells : $\Delta w_{c_j^v m}(t) = \alpha\ e_{s_{c_j^v}}(t)\ dS_{cm}^{jv}(t)$

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.

Blair, A. D. and Pollack, J. B. (1997). Analysis of dynamical recognizers. *Neural Computation*, 9(5):1127–1142.

Casey, M. P. (1996). The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6):1135–1178.

Das, S., Giles, C., and Sun, G. (1992). Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proceedings of The Fourteenth Annual Conference of the Cognitive Science Society*.

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

Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proc. IJCNN'2000, Int. Joint Conf. on Neural Networks*, Como, Italy.

Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with LSTM. In *Proc. ICANN'99, Int. Conf. on Artificial Neural Networks*, volume 2, pages 850–855, Edinburgh, Scotland. IEE, London.

Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*. To appear.

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

Kalinke, Y. and Lehmann, H. (1998). Computation in recurrent neural networks: From counters to iterated function systems. In *Advanced Topics in Artificial Intelligence*, volume 1502 of *LNAI*, Berlin,Heidelberg. Springer.

Lee, L. (1996). Learning of context-free languages: A survey of the literature. Technical Report TR-12-96, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts.

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.

Osborne, M. and Briscoe, E. (1997). Learning stochastic categorial grammars.

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

Plaut, D. C., Nowlan, S. J., and Hinton, G. E. (1986). Experiments on learning back propagation. Technical Report CMU–CS–86–126, Carnegie–Mellon University, Pittsburgh, PA.

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.

Rodriguez, P. and Wiles, J. (1998). Recurrent neural networks can learn to implement symbol-sensitive counting. In *Advances in Neural Information Processing Systems*, volume 10, pages 87–93. The MIT Press.

Rodriguez, P., Wiles, J., and Elman, J. (1999). A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40.

Siegelmann, H. T. (1993). Foundations of recurrent neural networks.

Siegelmann, H. T. and Sontag, E. D. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80.

Steijvers, M. and Grunwald, P. (1996). A recurrent network that performs a contextsensitive prediction task. In *Proceedings of the 18th Annual Conference of the Cognitive Science Society*. Erlbaum.

Sun, G. Z., Giles, C. L., Chen, H. H., and Lee, Y. C. (1993). The neural network pushdown automaton: Model, stack and learning simulations. Technical Report CS-TR-3118, University of Maryland, College Park.

Tonkes, B. and Wiles, J. (1997). Learning a context-free task with a recurrent neural network: An analysis of stability. In *Proceedings of the Fourth Biennial Conference of the Australasian Cognitive Science Society.*

Wiles, J. and Elman, J. (1995). Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *In Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages pages 482 – 487, Cambridge, MA. MIT Press.

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 Chauvin, Y. and Rumelhart, D. E., editors, *Back-propagation: Theory, Architectures and Applications*, chapter 13, pages 433–486. Hillsdale, NJ: Erlbaum.

Zeng, Z., Goodman, R., and Smyth, P. (1994). Discrete recurrent neural networks for grammatical inference. *IEEE Transactions on Neural Networks*, 5(2).