# DEKF-LSTM

Felix A. Gers, *gers@mantik.de*,
Neue Gruenstrasse 18, 10179 Berlin, Germany, *www.mantik.de*

Juan Antonio Pérez-Ortiz, *japerez@dlsi.ua.es*,
DLSI, Universitat d'Alacant, Spain, *www.dlsi.ua.es*

Douglas Eck *doug@idsia.ch*, Jürgen Schmidhuber *juergen@idsia.ch*,
IDSIA, Galleria 2, 6928 Manno, Switzerland, *www.idsia.ch*

**Abstract.**
Unlike traditional recurrent neural networks, the long short-term memory (LSTM) model generalizes well when presented with training sequences derived from regular and also simple nonregular languages. Our novel combination of LSTM and the decoupled extended Kalman filter, however, learns even faster and generalizes even better, requiring only the 10 shortest exemplars ($n \leq 10$) of the context sensitive language $a^n b^n c^n$ to deal correctly with values of $n$ up to 1000 and more. Even when we consider the relatively high update complexity per timestep, in many cases the hybrid offers faster learning than LSTM by itself.

## 1 Introduction

Sentences of regular languages are recognizable by finite state automata having obvious recurrent neural network (RNN) implementations. Most recent work on language learning with RNNs has focused on them. Only few authors have tried to teach RNNs to extract the rules of simple context free and context sensitive languages (CFLs and CSLs) whose recognition requires the functional equivalent of a potentially unlimited stack [11, 12, 1]. Some previous RNNs even failed to learn small CFL training sets [10]. Those that did not and those that even learned small CSL training sets [9, 1] failed to extract the general rules and did not generalize well on substantially larger test sets.

The recent *"Long Short-Term Memory"* (LSTM) method [6] is the first network that does not suffer from such generalization problems. It clearly outperforms traditional RNNs on all previous CFL and CSL benchmarks that we found in the literature. Stacks of potentially unlimited size are automatically and naturally implemented by linear units, the "Constant Error Carousels" (CECs) of standard LSTM, originally designed to overcome error decay problems plaguing previous RNNs [6]. Each linear CEC is surrounded by a few

nonlinear units responsible for learning the nonlinear aspects of sequence processing.

In this article we focus on improving LSTM convergence time by using a decoupled extended Kalman filter [8] to optimize learning rate. We consider two supervised learning algorithms: the original LSTM algorithm (discussed in next section), and a new one that combines the decoupled extended Kalman filter approach (DEKF-LSTM, described in Section 3). We apply both algorithms to the only CSL ever tried with RNNs, namely, $a^n b^n c^n$. Section 4 presents experiments; the results are discussed in Section 5.

## 2 LSTM overview

Unfortunately, lack of space prohibits a complete and self-contained description of LSTM. We refer the reader to [3, 4] for details. In what follows, we will limit ourselves to a brief overview. 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. 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 [3] 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 the rest of the units 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 such as "Real-Time Recurrent Learning" (RTRL), or "Back Propagation Through Time" (BPTT).

**Forward pass.** See [3] for a detailed description of LSTM's forward pass with forget gates and peephole connections. Essentially, the cell output $y^c$ is calculated based on the current cell state $s_c$ and four sources of input: to the cell itself, to the input gate, to the forget gate and input to output gate. All gates have a sigmoid squashing function with range $(0, 1)$. The state of memory cell $s_c(t)$ is calculated by adding the squashed gated input to the state at the previous time step $s_c(t-1)$, which is multiplied by the forget gate activation. The cell output $y^c$ is calculated by multiplying (gating) $s_c(t)$ by the output gate activation.

**Gradient-based backward pass.** Essentially, LSTM's backward pass (for details see [6, 4]) 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 $E(t)$, here the usual mean squared error function. Unlike BPTT and RTRL, LSTM's learning algorithm is local in space and time: the update complexity per time step and weight is $O(1)$. Still, LSTM learns many tasks unlearnable by BPTT and RTRL [6, 3, 4].

# 3 DEKF-LSTM overview

Due to lack of space, we provide only an overview of how DEKF is combined with LSTM. For an in-depth treatment of Kalman filters, see [8, 5]. Gradient descent algorithms, such as the original LSTM training algorithm, are usually slow when applied to time series because they depend on *instantaneous* estimations of the gradient: the derivatives of the objective function $E(t)$ only take into account the distance between the current output and the corresponding target, using no history information for weight updating. DEKF-LSTM overcomes this limitation. It considers training as an optimal filtering problem, recursively and efficiently computing a solution to the least-squares problem: finding the curve of best fit for a given set of data in terms of minimizing the average distance between data and curve. At any given time step, all the information supplied to the network up until now is used, including all derivatives computed since the first iteration of the learning process. However, computation is done such that only the results from the previous step need to be stored.

The Kalman filter requires, among other things, the computation of the derivatives of the objective function $E(t)$. In our implementation these are calculated in the same way as in the original LSTM backward pass. In addition, at every discrete time step of each training sequence, several matrix operations are performed by DEKF-LSTM, including the inversion of a square matrix of size equal to the number of output neurons. Therefore, while LSTM is local in time and space, DEKF-LSTM is not.

# 4 Experiments

The network sequentially observes exemplary symbol strings of a given language (in this case the CSL $a^n b^n c^n$), 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 next symbol, 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. 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 "end of string" 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.

## 4.1 Network topology and 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. 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 $(-2, 2)$.

We use a network with 4 input and output units, and two memory blocks (with one cell each), resulting 84 adjustable weights (72 unit-to-unit and 12 bias connections).

## 4.2 Training and testing

Training 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: $3n$ for $a^n b^n c^n$. Only positive exemplars are presented. Training is stopped once all training sequences have been accepted. All results are averages over 10 independently trained networks with different weight initializations (the same for each experiment). The *generalization set* is the largest accepted test set.

We study LSTM's behavior in response to two kinds of training sets: a) with $n \in \{1,..,N\}$ (we focus on $N = 10$) and b) with $n \in \{N-1, N\}$ (we focus on $N = 21$). For large values of $N$, case (b) is much harder because there is no support from short (and easier to learn) strings. We test all sets with $n \in \{L,..,M\}$, $L \in \{1,..,N-1\}$.

**LSTM weight updating.** Weight changes are made after each sequence. We apply either a constant learning rate or the momentum algorithm [7] with momentum parameter 0.99. At most $10^7$ training sequences are presented; we test with $M \in \{N,..,500\}$ (sequences of length $\leq 1500$).

**DEKF-LSTM weight updating.** The *online* nature of the DEKF-LSTM algorithm forces weights to be updated after each symbol presentation. The pa-

rameters of the algorithm are set as follows (see [5] for details): the covariance matrix of the measurement noise is annealed from 100 to 1; the covariance matrix of artificial process noise is set to 0.005 (unless specified otherwise). These values gave good results in preliminary experiments, but they are not critical and there is a big range of values which result in similar learning performance. The influence of the remaining parameter, the initial error covariance matrix, will be studied in Section 5.2. The maximum of training sequences presented is $10^2$; we test with $M \in \{N,..,10000\}$ (sequences of length $\leq 30000$).

# 5 Results

**Previous results:** Chalup and Blair [2] reported that a simple recurrent network trained with a hill-climbing algorithm can learn the training set for $n \leq 12$, but they did not give generalization results. Boden and Wiles [1] trained a sequential cascaded network with BPTT; for a training set with $n \in \{1,..,10\}$, the best networks generalized to $n \in \{1,..,12\}$ in 8% of the trials.

## 5.1 LSTM with constant learning rate or momentum

When utilizing the original training algorithm, LSTM learns both training sets and generalizes well. With a training set with $n \in \{1,..,10\}$ the best generalization was $n \in \{1,..,52\}$ (the average generalization was $n \in \{1,..,28\}$). A training set with $n \in \{1,..,40\}$ was sufficient for perfect generalization up to the tested maximum: $n \in \{1,..,500\}$ (sequences of length up to 1500).

LSTM worked well for a wide range of learning rates (about three orders of magnitude) — see Table 1. Use of the momentum algorithm [7] clearly helped to improve learning speed (allowing the same range for the initial learning rate). The choice of learning rate did not affect generalization performance (not reported in Table 1).

## 5.2 DEKF-LSTM results

The DEKF-LSTM combination significantly improves the LSTM results. Very small training sets with $n \in \{1,..,10\}$ are sufficient for perfect generalization up to values of $n \in \{1,..,2000\}$ and more: one of the experiments with $\delta = 10^2$ gave a generalization set with $n \in \{1,..,10000\}$. We ask the reader to briefly reflect on what this means: after a short training phase the system worked so robustly and precisely that it saw the difference between, say, $a^{8888}b^{8888}c^{8888}$ and $a^{8888}b^{8888}c^{8889}$!

With training set $n \in \{1,..,10\}$ and $\delta = 10$ the average generalization set was $n \in \{1,..,434\}$ (the best generalization was $n \in \{1,..,2743\}$), whereas with the original training algorithm it was $n \in \{1,..,28\}$. What is more, training is usually completed after only $2 \cdot 10^3$ training strings, whereas the original algorithm needs a much larger number of strings.

Table 1: results for CSL $a^n b^n c^n$ for training sets with $n$ ranging from 1 to 10 and from 20 to 21, with various (initial) learning rates ($10^{-\alpha}$) with and without momentum (momentum parameter 0.99). Showing (from left to right, for each set with and without momentum): the average number of training sequences and the percentage of correct solutions once the training set was learned.

| | (1,..,10) | | | | (20,21) | | | |
|---|---|---|---|---|---|---|---|---|
| | Momentum | | Constant | | Momentum | | Constant | |
| $\alpha$ | Train Seq [$10^3$] | % Corr | Train Seq [$10^3$] | % Corr | Train Seq [$10^3$] | % Corr | Train Seq [$10^3$] | % Corr |
| 1 | - | 0 | - | 0 | - | 0 | - | 0 |
| 2 | - | 0 | - | 0 | - | 0 | - | 0 |
| 3 | - | 0 | 68 | 100 | - | 0 | 1170 | 30 |
| 4 | 20 | 90 | 351 | 100 | - | 0 | 7450 | 30 |
| 5 | 45 | 90 | 3562 | 100 | 127 | 20 | 1205 | 20 |
| 6 | 329 | 100 | - | 0 | 1506 | 20 | - | 0 |
| 7 | 3036 | 100 | - | 0 | 1366 | 10 | - | 0 |

Table 2 shows the influence of the parameter $\delta$, which is used to determine the initial error covariance matrix in the Kalman filter. The rest of the parameters are set as indicated before, except for the covariance matrix of artificial process noise which is annealed from 0.005 to $10^{-6}$ for the training set with $n$ being either 20 or 21.

We observe that learning speed and accuracy (percentage of correct solutions) are considerably improved (compare Table 1). The number of training sequences is considerably smaller, and the percentage of successful solutions in the case of $(20, 21)$ is far greater.

However, DEKF-LSTM's computational complexity per time step and weight is much larger than LSTM's. To account for this we derived a relative CPU time unit that corresponds to computation time for one epoch (i.e., 1000 sequence presentations) of LSTM training. This relative CPU time is shown for DEKF-LSTM in parentheses in Table 2 and can be compared directly to "number of training sequence" values in Table 1.

A comparision of LSTM and DEKF-LSTM using this relative measure reveals that the additional complexity of DEKF-LSTM is largely compensated for by the smaller number of training sequences needed for learning the training set. Compare, for example, the (20,21) case. DEKF-LSTM with $\delta = 10^{-2}$ achieves 90% correct solutions in 84 relative CPU units. This compares favoribly with LSTM performance (see Table 1 for LSTM figures).

A lesser problem of DEKF-LSTM is its occasional instability. Learning usually takes place in the beginning of the training phase or never at all. All failures in Table 2 are due to this.

Table 2: results for CSL $a^n b^n c^n$ for training sets with $n$ ranging from 1 to 10 and from 20 to 21, using DEKF-LSTM with different initial values for elements of the error covariance matrix, $\delta^{-1}$. Showing (from left to right, for each set): the average number of training sequences (CPU time in relative units given in parenthesis, see text for details) and the percentage of correct solutions until training set was learned.

| | (1,.., 10) | | (20, 21) | |
|---|---|---|---|---|
| $\delta = 10^b$ with $b =$ | Train Seq $[10^3]$ | % Corr | Train Seq $[10^3]$ | % Corr |
| $-3$ | 2 (46) | 20 | - | 0 |
| $-2$ | 2 (46) | 80 | 4 (84) | 90 |
| $-1$ | 2 (46) | 100 | 4 (84) | 70 |
| 0 | 2 (46) | 60 | 8 (168) | 70 |
| 1 | 2 (46) | 100 | 12 (252) | 60 |
| 2 | 2 (46) | 70 | 4 (84) | 50 |
| 3 | 2 (46) | 80 | 5 (105) | 50 |

### 5.3 Analysis of the network solution

How does LSTM solve the CSL $a^n b^n c^n$ problem? With both training approaches, the network uses, in general, a combination of two counters, instantiated separately in the two memory blocks. For example one counter would increase on the symbol $a$ and then decrease on the symbol $b$. By counting up with a slightly lower stepsize than it counts down, such a device can identify when an equal number of $a$ and $b$ symbols have been presented. At any time the occurrence of a $c$ symbol would cause the block to close its input input gate and open its forget gate, emptying cell contents. A second counter would do the same thing for symbols $b$, $c$, and $a$, respectively. In this case an equal number of $b$ and $c$ symbols would bring about the prediction of sequence terminator $T$. In short, one memory block solves $a^n b^n$ while another solves $b^n c^n$. By working together they are able to solve the much more difficult CSL task. All of this works in extremely precise and robust fashion — otherwise it would be impossible to separate strings such as $a^{1000} b^{1000} c^{1000}$ and $a^{1000} b^{999} c^{1000}$.

## 6 Conclusion

LSTM is the first RNN to generalize well on non-regular language benchmarks. But by combining LSTM and the decoupled extended Kalman filter DEKF we obtain a system that needs orders of magnitude fewer training sequences and generalizes even better than standard LSTM. The hybrid requires only training exemplars shorter than $a^{11} b^{11} c^{11}$ to extract the general rule of the

context sensitive language $a^n b^n c^n$ and to generalize correctly for all sequences up to $a^{1000} b^{1000} c^{1000}$ and beyond.

We also verified that DEKF-LSTM is not outperformed by LSTM on other traditional benchmarks involving continuous data, where LSTM outperformed traditional RNNs [6, 4]. So DEKF-LSTM is not overspecialized on CSLs but represents a general advance. The update complexity per training example, however, is worse than LSTM's, which is local in time and space.

**Acknowledgments.** This work was supported by Mantik GmbH (Berlin), SNF grant 2100-49'144.96, "Long Short-Term Memory", and Generalitat Valenciana grant FPI-99-14-268.

# References

[1] M. Boden and J. Wiles. Context-free and context-sensitive dynamics in recurrent neural networks, 2000.

[2] S. Chalup and A. Blair. Hill climbing in recurrent neural networks for learning the $a^n b^n c^n$ language. In *Proceedings of the 6th Confernce on Neural Information Processing*, pages 508–513, Perth, 1999.

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

[4] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[5] Simon Haykin. *Neural networks: a comprehensive foundation.* Prentice-Hall, New Jersey, 2nd edition, 1999.

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

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

[8] G. V. Puskorius and L. A. Feldkamp. Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. *IEEE Transactions on Neural Networks*, 5(2):279–297, 1994.

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

[10] Paul Rodriguez and Janet Wiles. 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, 1998.

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

[12] J. Wiles and J. Elman. 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, 1995. MIT Press.