Learning Context Sensitive Languages with LSTM Trained with Kalman Filters*

Felix A. Gers[1], Juan Antonio Pérez-Ortiz[2], Douglas Eck[3], and Jürgen Schmidhuber[3]

[1] Mantik Bioinformatik GmbH, Neue Gruenstrasse 18, 10179 Berlin, Germany
[2] DLSI, Universitat d'Alacant, E-03071 Alacant, Spain
[3] IDSIA, Galleria 2, 6928 Manno, Switzerland

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. Some previous RNNs even failed to learn small CFL training sets [9]. Others succeeded at CFL and even small CSL training sets [8,1], but 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 [4]. 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 called *constant error carousels* (CECs).

In this article we focus on improving LSTM convergence time by using the decoupled extended Kalman filter (DEKF) [7] learning algorithm. We compare DEKF with the original gradient-descent algorithm when applied to the only CSL ever tried with RNNs, namely, $a^n b^n c^n$.

* Work supported by SNF grant 2100-49'144.96, Spanish Comisión Interministerial de Ciencia y Tecnología grant TIC2000-1599-C02-02, and Generalitat Valenciana grant FPI-99-14-268.

Gers, F.A., Pérez-Ortiz, J.A., Eck, D., Schmidhuber, J. (2002). Learning Context Sensitive Languages with LSTM Trained with Kalman Filters. In: Dorronsoro, J.R. (eds) Artificial Neural Networks — ICANN 2002. ICANN 2002. Lecture Notes in Computer Science, vol 2415. Springer, Berlin, Heidelberg.

2 Architecture and learning algorithms

Lack of space prohibits a complete description of LSTM [3,6]. 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 flow "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 no longer useful (*forget gate*); peephole connections [4] allow gates direct access to the CEC states.

Output function derivatives in LSTM are computed in such a way as to permit errors to flow untruncated only through the CECs. The CECs use their linear counters to find long temporal dependencies in the signal, thus freeing the rest of the network to learn other nonlinear aspects of the signal. The resulting architecture is able to find relevant information separated by very long time lags, allowing it to succeed where traditional RNNs fail.

Kalman Filters. All previous works on LSTM considered gradient descent as learning algorithm, minimizing the usual mean squared error function. When using gradient descent, the update complexity per time step and weight is $O(1)$. 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 [7,5]. 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 only take into account the distance between the current output and the corresponding target, using no history information for weight updating. DEKF overcomes this limitation. It considers training as an optimal filtering problem, recursively and efficiently computing a solution to the least-squares problem. At any given time step, all the information supplied to the network up until now is used, but only the results from the previous step need to be stored. DEKF requires, among other things, the computation of the output function derivatives, which are calculated in the same way as in the gradient-descent LSTM. In addition, at every discrete time step several matrix operations are performed, including the inversion of a square matrix of size equal to the number of output neurons. Therefore, while original LSTM is local in time and space, the combination DEKF-LSTM is not.

3 Experiments

The network sequentially observes exemplary symbol strings of the CSL $a^n b^n c^n$ presented one 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. A string is accepted when all predictions have been correct; otherwise it is rejected. A system has learned a given language up to string size n once it is able to correctly predict all strings with size $\leq n$.

Network Topology and Parameters. The input units are fully connected to a hidden layer consisting of 2 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. The bias weights to input gate, forget gate and output gate are initialized with -1.0, $+2.0$ and -2.0, respectively (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 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: symbols are encoded locally; therefore a unit is needed for each symbol of the alphabet, and one additional unit is required for T. $+1$ signifies that a symbol is set and -1 that it is not set; the decision boundary for the network output is 0.0.

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 length $3n$ (only positive exemplars). 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 $n \in \{1,..,N\}$ and b) with $n \in \{N-1, N\}$; we focus on $N = 10$ for the first case and on $N = 21$ for the second one. 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 $n \in \{L,..,M\}$ and $L \in \{1,..,N-1\}$, where M will be specified later.

In the case of gradient descent, weight changes are made after each sequence. We apply either a constant learning rate or the momentum algorithm with momentum parameter 0.99. At most 10^7 training sequences are presented; we test with $M \in \{N,..,500\}$ (sequences of length ≤ 1500).

On the other hand, when using Kalman filters, the online nature of the basic DEKF algorithm forces weights to be updated after each symbol presentation. The parameters of the algorithm are set as follows (see [5] for details): the covariance matrix of the measurement noise is annealed from 100 to 1, and 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 later. The maximum of training sequences presented with DEKF is 10^2; we test with $M \in \{N,..,10000\}$ (sequences of length ≤ 30000).

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. Showing (from left to right for each set): 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	
α	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

4 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.

Gradient-Descent Results. 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: $n \in \{1,..,500\}$.

LSTM worked well for a wide range of learning rates (about three orders of magnitude) — see Table 1. Use of the momentum algorithm clearly helped to improve learning speed (allowing the same range for the initial learning rate).

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 ($\delta = 10^2$) 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, sequences $a^{8888}b^{8888}c^{8888}$ 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

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 with different initial values for elements of the error covariance matrix, δ^{-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

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 2 shows the influence of the parameter δ, 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 much 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 original 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 comparison 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 with $\delta = 10^{-2}$ achieves 90% correct solutions in 84 relative CPU units. This compares favorably 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.

Analysis of the Network Solution. 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 gate and open its forget gate, emptying cell contents. A second counter would do the same thing for symbols b and c. 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 solve the much more difficult CSL task.

5 Conclusion

LSTM is the first RNN to generalize well on non-regular language benchmarks. But by combining LSTM and 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 CSL $a^n b^n c^n$ and to generalize correctly for all sequences up to $n = 1000$ 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 [6,3]. This indicates that DEKF-LSTM is not over-specialized on CSLs but represents a general advance. The update complexity per training example, however, is worse than LSTM's.

References

1. Boden, M., Wiles, J.: Context-free and context-sensitive dynamics in recurrent neural networks. Connection Science **12**, 3 (2000).
2. Chalup, S., Blair, A.: Hill climbing in recurrent neural networks for learning the $a^n b^n c^n$ language. Proc. 6th Conf. on Neural Information Processing (1999) 508–513.
3. Gers, F. A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM. Neural Computation **12**, 10 (2000) 2451–2471.
4. Gers, F. A., Schmidhuber, J.: LSTM recurrent networks learn simple context free and context sensitive languages. IEEE Transactions on Neural Networks **12**, 6 (2001) 1333–1340.
5. Haykin, S. (ed.): Kalman filtering and neural networks. Wiley (2001).
6. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**, 8 (1997) 1735–1780.
7. Puskorius, G. V., Feldkamp, L. A.: Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. IEEE Transactions on Neural Networks **5**, 2 (1994) 279–297.
8. Rodriguez, P., Wiles, J., Elman, J.: A recurrent neural network that learns to count. Connection Science **11**, 1 (1999) 5–40.
9. Rodriguez, P., Wiles, J.: Recurrent neural networks can learn to implement symbol-sensitive counting. Advances in Neural Information Processing Systems **10** (1998) 87–93. The MIT Press.