

Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets*

Juan Antonio Pérez-Ortiz,[†] Felix A. Gers,[‡] Douglas Eck,[§] and Jürgen Schmidhuber[§]

Abstract

The Long Short-Term Memory (LSTM) network trained by gradient descent solves difficult problems which traditional recurrent neural networks in general cannot. We have recently observed that the decoupled extended Kalman filter training algorithm allows for even better performance, reducing significantly the number of training steps when compared to the original gradient descent training algorithm. In this paper we present a set of experiments which are unsolvable by classical recurrent networks but which are solved elegantly and robustly and quickly by LSTM combined with Kalman filters.

1 Introduction

The decoupled extended Kalman filter (DEKF) (Puskorius and Feldkamp, 1994; Haykin, 2001) has been used successfully to optimize the training of recurrent neural networks (RNNs). Typically DEKF requires fewer training steps and yields better generalization than the usual gradient descent algorithms. In the DEKF framework, learning is treated as a filtering problem in which the optimum weights of the network are estimated efficiently in a recursive fashion. The algorithm is especially suitable for online learning situations, where weights are adjusted continuously, although it can be applied to batch training as well (Feldkamp and Puskorius, 1994).

With DEKF it should be possible for a RNN to learn optimal weights for many difficult problems. In principle, a RNN (Elman, 1990; Robinson and Fallside, 1991; Williams and Zipser, 1992; Schmidhuber, 1992; Pearlmutter, 1995) is able to instantiate arbitrary temporal dynamics. This suggests great potential for solving hard problems that cannot be dealt with using feedforward networks. In practice, however, RNNs are unable to bridge long time lags between important events: errors flowing backward in time either decay exponentially or blow up (Hochreiter et al., 2001a). These so-called *vanishing gradients* limit traditional RNNs to problems having only short time lags (where one could also use feedforward networks instead of RNNs). A recent novel RNN called *Long Short-Term Memory* (LSTM) (Hochreiter and Schmidhuber, 1997) overcomes this fundamental long-term dependency problem by enforcing constant error flow. That's why LSTM learns previously unlearnable solutions to numerous tasks (Hochreiter and Schmidhuber, 1997; Gers et al., 2000; Gers and Schmidhuber, 2001), including tasks that require storing relevant events for more than 1000 subsequent discrete time steps without the help of any short training sequences. LSTM's applicability is broad. For

**Neural Networks* 16(2), 2003. This work was supported by the Generalitat Valenciana grant FPI-99-14-268, by the Spanish Comisión Interministerial de Ciencia y Tecnología grant TIC2000-1599-C02-02, and by the Swiss National Foundation grant 2100-49'144.96.

[†]Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, E-03071 Alacant, Spain

[‡]Mantik Bioinformatik GmbH, Neue Gruenstrasse 18, 10179 Berlin, Germany

[§]IDSIA, Galleria 2, 6913 Manno-Lugano, Switzerland

instance, it has recently been employed as a powerful function approximator for reinforcement learning (RL) in partially observable environments (Bakker, 2001). In situations where the optimal next action depends on long-term memory of old sensory inputs, traditional RL fails while RL with LSTM function approximation succeeds. LSTM also builds the basis of the first working gradient-based metalearner (Hochreiter et al., 2001b) — a very recent machine that can *learn* fast learning algorithms for nontrivial classes of functions.

All previous LSTM implementations have used a form of gradient descent to adjust the weights of the network. In this paper we apply the DEKF training algorithm to the LSTM architecture for the first time; we compare experimental results obtained with the gradient descent algorithm to those of DEKF, and also comment on much worse results obtained with traditional RNNs.

Section 2 introduces the LSTM architecture. Section 3 describes the learning algorithms to be considered in the paper, namely, gradient descent and DEKF. Next, in Section 4 the experiments are discussed in detail; two different experiments are presented: online prediction over symbolic sequences with long-term dependencies, and inference of a context sensitive language. Finally, the paper concludes with a short discussion.

2 Long Short-Term Memory

A main distinction between LSTM (Hochreiter and Schmidhuber, 1997) and traditional RNNs (Elman, 1990; Robinson and Fallside, 1991; Pearlmutter, 1995) is LSTM’s use of a memory cell containing a self-connected linear unit. This so-called *constant error carousel* (CEC) enforces constant error flow and overcomes a fundamental problem plaguing previous RNNs: it prevents error signals from decaying quickly as they flow *back in time*. At any point in time, errors in the network (whether from cells or from *gates*, described below) are used to drive weight changes. However, only the CECs keep track of error as it flows back in time; errors elsewhere are truncated (errors outside CECs vanish exponentially fast anyway, just like in traditional RNNs). By tracking long-timescale dependencies in the CECs, LSTM is able 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, even with complex update algorithms such as real-time recurrent learning (Williams and Zipser, 1989; Robinson and Fallside, 1991) (RTRL) or backpropagation through time (Williams and Peng, 1990) (BPTT). But note that even time window approaches based on feedforward networks can deal with 10 step time lags!

In general, however, it is not sufficient to simply add linear counters to a RNN. Without some method of protecting the contents of these counters, such a network could quickly diverge. For this reason, LSTM CECs are arranged in *memory blocks* of cells that control the flow of information through the CECs. Though LSTM could in principle work with any differentiable protective mechanism, existing implementations use a small set of multiplicative gates: an *input gate* learns to protect the CECs from irrelevant inputs, an *output gate* learns to turn off a cell block that is generating irrelevant output, and a *forget gate* allows CECs to reset themselves to zero when necessary. See (Gers et al., 2000) for a detailed description of LSTM with input, output and forget gates. Figure 1 shows a memory block with a single cell; this figure will prove useful for understanding the notation to be used throughout the paper. Memory blocks form the hidden layer of a LSTM network as shown in Figure 2, biases are not shown.

Following is a brief description of the notation to be used throughout the paper. We denote with n_U , n_Y , n_M and n_C the number of input neurons, output neurons, memory blocks and cells, respectively. The input at time step t is denoted with vector $\mathbf{u}(t)$ and the corresponding output of the network with $\mathbf{y}(t)$. The v -th cell in the i -th block is represented by c_i^v .

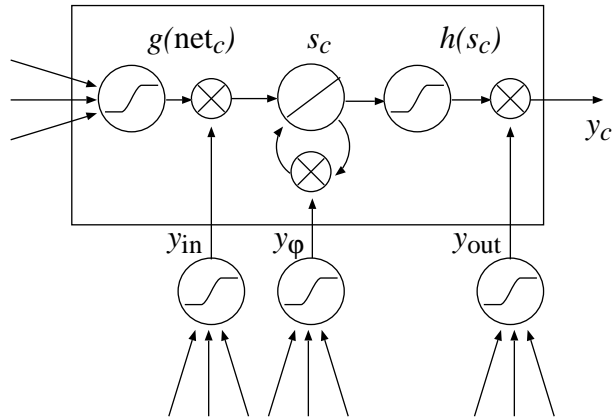


Figure 1: A LSTM memory block with one cell. If the memory block had two or more cells, the three gates (below; from left to right: input gate, forget gate, and output gate) would be shared by all of them.

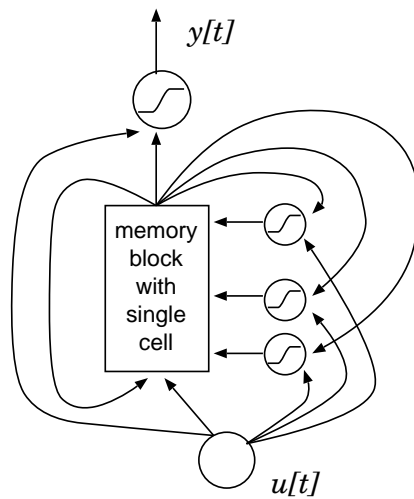


Figure 2: A single-input single-output three-layer LSTM architecture with only one memory block with one single cell within ($n_M = n_C = 1$). All the connections except for the biases are shown.

When representing the weights, superscripts indicate the computation in which the weight is involved: the “ φ, c ” in $W_{j,c_i}^{\varphi,c}$ indicates that the weight is used to compute the activation of a forget gate (φ) from a cell (c); the “out” in W_j^{out} (a bias) indicates that it is used to compute an output gate. Subscripts designate the particular units involved and run parallel to superscripts.

2.1 Forward pass

The activation of the input gate in the j th memory block, y_{in_j} , is computed as:

$$\text{net}_{\text{in}_j}(t) = \sum_{i=1}^{n_M} \sum_{v=1}^{n_C} W_{j,c_i}^{\text{in},c} y_{c_i^v}(t-1) + \sum_{m=1}^{n_U} W_{j,m}^{\text{in},u} u_m(t) + W_j^{\text{in}} \quad (1)$$

$$y_{\text{in}_j}(t) = \theta(\text{net}_{\text{in}_j}(t)) \quad (2)$$

where θ is the squashing function for the gates in the network (usually a logistic sigmoid function with range (0,1)). Similarly, forget gate activation y_φ is obtained by computing:

$$\text{net}_{\varphi_j}(t) = \sum_{i=1}^{n_M} \sum_{v=1}^{n_C} W_{j,c_i}^{\varphi,c} y_{c_i^v}(t-1) + \sum_{m=1}^{n_U} W_{j,m}^{\varphi,u} u_m(t) + W_j^\varphi \quad (3)$$

$$y_{\varphi_j}(t) = \theta(\text{net}_{\varphi_j}(t)) \quad (4)$$

The weights involved in the forget gates are usually initialized so that $\mathbf{y}_\varphi(0)$ is close to $\mathbf{1}$; with this initialization, cells do not forget anything until they learn how to forget. The output gate activation \mathbf{y}_{out} is computed as follows:

$$\text{net}_{\text{out}_j}(t) = \sum_{i=1}^{n_M} \sum_{v=1}^{n_C} W_{j,c_i}^{\text{out},c} y_{c_i^v}(t-1) + \sum_{m=1}^{n_U} W_{j,m}^{\text{out},u} u_m(t) + W_j^{\text{out}} \quad (5)$$

$$y_{\text{out}_j}(t) = \theta(\text{net}_{\text{out}_j}(t)) \quad (6)$$

The internal state of memory cell \mathbf{s}_c is calculated by adding the squashed gated input to the state at time $t-1$ multiplied by the corresponding forget gate:

$$s_{c_j^v}(t) = y_{\varphi_j}(t) s_{c_j^v}(t-1) + y_{\text{in}_j}(t) g(\text{net}_{c_j^v}(t)) \quad (7)$$

where g is a squashing function and

$$\text{net}_{c_j^v}(t) = \sum_{i=1}^{n_M} \sum_{k=1}^{n_C} W_{c_j^v,c_i^k}^{c,c} y_{c_i^k}(t-1) + \sum_{m=1}^{n_U} W_{c_j^v,m}^{c,u} u_m(t) + W_{c_j^v}^c \quad (8)$$

with $s_{c_j^v}(0) = 0$ for all j and v . The cell output \mathbf{y}_c is computed by squashing the cell state and gating (multiplying) it by the activation of the output gate:

$$y_{c_j^v}(t) = y_{\text{out}_j}(t) h(s_{c_j^v}(t)) \quad (9)$$

where h is a squashing function. Finally, if we allow direct shortcut connections between input and output layers, the global output of the network \mathbf{y} is calculated by:

$$\text{net}_k(t) = \sum_{j=1}^{n_M} \sum_{v=1}^{n_C} W_{k,c_j^v}^{y,c} y_{c_j^v}(t) + \sum_{m=1}^{n_U} W_{k,m}^{y,u} u_m(t) + W_k^y \quad (10)$$

$$y_k(t) = f(\text{net}_k(t)) \quad (11)$$

where f is again a suitable squashing function.

Recently *peephole connections* were added to the LSTM model (Gers and Schmidhuber, 2001). Peephole connections overcome a limitation of traditional LSTM, namely, the lack of direct connections from the CEC to the gates, which are supposed to control it. This new set of weighted connections links each CEC to the corresponding gates. Some of the experiments in Section 4 will use LSTM networks with peephole connections.

2.2 Gradient computation in the backward pass

In the backward pass the partial derivatives of the overall nonlinearity $\mathbf{y}(t)$ with respect to each weight are calculated. These derivatives are $\partial y_i(t)/\partial W_j(t)$ with $i = 1, \dots, n_Y$ and W_j denoting a generic synaptic weight.

Lack of space prohibits a complete and self-contained description of the way of computing the partial derivatives for LSTM (we refer the reader to the paper by Gers et al. (2000) for details). Essentially, it is an efficient fusion of slightly modified, truncated BPTT and a customized version of RTRL.

Truncation ensures that errors cannot re-enter cells. This in turn ensures constant error flow through the CEC in the memory cell avoiding the vanishing gradient problem: the influence of an earlier event on the current output of a traditional RNN decreases exponentially with time.

Consider a simple single-input single-output LSTM network, like that in Figure 2, with just one cell state s ; by following the chain rule, the influence of input $u(1)$ on current output $y(t)$ may be written as:

$$\frac{\partial y(t)}{\partial u(1)} = \frac{\partial y(t)}{\partial s(t)} \prod_{l=2}^t \frac{\partial s(l)}{\partial s(l-1)} \frac{\partial s(1)}{\partial u(1)} \quad (12)$$

The vanishing gradient problem happens due to the fact that the product in (12) goes to zero very quickly (Hochreiter et al., 2001a), that is, the error *vanishes*, and nothing can be learned in acceptable time. The way in which gradient is computed in LSTM architecture surpasses the problem by enforcing

$$\frac{\partial s(l)}{\partial s(l-1)} = 1 \quad (13)$$

for all l . Therefore, the error flow within the CECs is constant, and long-term dependencies may be taken into account. Equation (13) represents the key to LSTM's success (Hochreiter and Schmidhuber, 1997). Most LSTM networks implement (13) with a linear unit having a 1.0 self connection, but other alternatives may also be of interest. The nature of the nonlinear gates surrounding and protecting the CECs may vary as well.

3 Training algorithms

3.1 Gradient descent

The gradient descent algorithm applies a correction $\Delta W_j(t)$ to the weight $W_j(t)$ which is proportional to the partial derivative $\partial E(t)/\partial W_j(t)$, where E is the objective function, here the usual quadratic error function,

$$E = \frac{1}{2} \sum_{i=1}^{n_Y} (d_i(t) - y_i(t))^2 \quad (14)$$

and $d_i(t)$ is the target or desired response for the i th output neuron at time t . The contribution at time t to W_j update is defined by

$$\Delta W_j(t) = -\alpha \frac{\partial E(t)}{\partial W_j(t)} = \alpha \sum_{i=1}^{n_Y} \frac{\partial y_i(t)}{\partial W_j(t)} \quad (15)$$

where α is the learning rate parameter.

When applied to LSTM, gradient descent uses not more than $O(1)$ computations per connection and time step, thus being local in space and time (Schmidhuber, 1989). All previous papers concerning LSTM used the gradient descent learning algorithm; in this paper we consider for the first time the DEKF training algorithm as well.

3.2 Kalman filters

Gradient descent algorithms, such as the original LSTM training algorithm, are usually slower than necessary when applied to time series because they depend on *instantaneous* estimations of the gradient: the derivatives of the error function with respect to the weights to be adjusted only take into account the distance between the current output and the corresponding target, using no history information for weight updating.

The DEKF (Puskorius and Feldkamp, 1994; Haykin, 1999) overcomes this limitation. It considers training as an optimal filtering problem, recursively and efficiently computing a solution to the least-squares problem of 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.

Lack of space prohibits a complete description of DEKF. We refer the reader to previous citations for details. In what follows, we will limit ourselves to a brief overview. The extended Kalman filter is used for training neural networks (recurrent or not) by assuming that the optimum setting of the weights is stationary. However, when considering all the weights of the network together, the resulting matrices become so unmanageable (even for networks with moderate sizes) that a *node-decoupled* version of the algorithm is usually used instead to make the problem computationally tractable. The decoupled approach applies the extended Kalman filter independently to each neuron in order to estimate the optimum weights feeding it. By proceeding this way, only local interdependencies are considered. The equations for iteration t of a DEKF minimizing the typical quadratic error measure in (14) can be formulated as follows:

$$\mathbf{G}_i(t) = \mathbf{K}_i(t-1)\mathbf{C}_i^T(t) \left[\sum_{i=1}^g \mathbf{C}_i(t)\mathbf{K}_i(t-1)\mathbf{C}_i^T(t) + \mathbf{R}(t) \right]^{-1} \quad (16)$$

$$\mathbf{w}_i(t) = \mathbf{w}_i(t-1) + \mathbf{G}_i(t) [\mathbf{d}(t) - \mathbf{y}(t)] \quad (17)$$

$$\mathbf{K}_i(t) = \mathbf{K}_i(t-1) - \mathbf{G}_i(t)\mathbf{C}_i(t)\mathbf{K}_i(t-1) + \mathbf{Q}_i(t) \quad (18)$$

where g is the number of neurons, i defines a particular neuron ($1 \leq i \leq g$), and \mathbf{w}_i is a vector with all the weights leading to neuron i . The difference in (17) between the desired response $\mathbf{d}(t)$ and the actual output vector of the network $\mathbf{y}(t)$ defines what is known as *innovation*.

Let n_i denote the number of weights leading to neuron i . Matrix \mathbf{G}_i computed in (16) is an $n_i \times n_Y$ matrix denoting the Kalman *gain* for neuron i . Matrix \mathbf{K}_i computed in (18) is an $n_i \times n_i$ matrix denoting the *error covariance matrix* for neuron i . Matrix \mathbf{Q}_i is the covariance matrix of *artificial process noise* for neuron i and is used to overcome some divergence problems of the filter. Matrix \mathbf{R} is the covariance matrix of the *measurement noise*. Matrices \mathbf{K} , \mathbf{Q} and \mathbf{R} are initialized in a problem-specific manner. Finally, the Jacobian \mathbf{C}_i is an $n_Y \times n_i$ matrix containing the partial derivatives of the function defining the output \mathbf{y} of the network with respect to each weight leading to neuron i :

$$\mathbf{C}_i(t) = \begin{pmatrix} \frac{\partial y_1}{\partial w_{i1}}(t) & \frac{\partial y_1}{\partial w_{i2}}(t) & \dots & \frac{\partial y_1}{\partial w_{in_i}}(t) \\ \frac{\partial y_2}{\partial w_{i1}}(t) & \frac{\partial y_2}{\partial w_{i2}}(t) & \dots & \frac{\partial y_2}{\partial w_{in_i}}(t) \\ \dots & \dots & \dots & \dots \\ \frac{\partial y_{n_Y}}{\partial w_{i1}}(t) & \frac{\partial y_{n_Y}}{\partial w_{i2}}(t) & \dots & \frac{\partial y_{n_Y}}{\partial w_{in_i}}(t) \end{pmatrix} \quad (19)$$

Combining DEKF with the LSTM architecture is straightforward. We consider a group of weights for each neuron in LSTM, that is, a group for each different gate, cell and output neuron, giving $g = n_M(n_C + 3) + n_Y$. At time step t we calculate the derivatives required for matrix $\mathbf{C}_i(t)$ as indicated in 2.2, and then apply equations (16)–(18) in order to update weights $\mathbf{w}_i(t)$.

It should be noted that DEKF’s time complexity (Haykin, 1999, p. 771) is much larger than that of the gradient descent algorithm because DEKF not only has to compute the same derivatives, but also many matrix operations (including the inversion of the square matrix in (16) of size $n_Y \times n_Y$) at every time step. Thus, while original gradient descent LSTM consumes $O(1)$ resources per connection and time step (Schmidhuber, 1989), DEKF is not local in time and space.

4 Experiments

We study LSTM’s performance with both learning algorithms, gradient descent and DEKF, on two tasks that are unsolvable by traditional RNNs (Elman, 1990; Robinson and Fallside, 1991; Pearlmutter, 1995).

Online learning. See Section 4.1. The next symbol of a symbolic continual input stream generated from the difficult embedded Reber automaton has to be predicted. The focus is on online learning, where the network is forced to give in real time an output as correct as possible for the input supplied at each time step. We investigate the number of symbols needed to attain correct predictions during a large period of time.

Context sensitive language learning. See Section 4.2. Sentences of regular languages are recognizable by finite state automata having obvious 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 (Sun et al., 1993; Wiles and Elman, 1995; Boden and Wiles, 2000). Some previous RNNs even failed to learn small CFL training sets (Rodriguez and Wiles, 1998). Those that did not and those that even learned small CSL training sets (Rodriguez et al., 1999; Boden and Wiles, 2000) failed to extract the general rules and did not generalize well on substantially larger test sets.

LSTM is the first network that does not suffer from such generalization problems (Gers and Schmidhuber, 2001). It clearly outperforms traditional RNNs on all previous CFL and CSL benchmarks found in the literature. Stacks of potentially unlimited size are automatically and naturally implemented by the CECs.

Here we concentrate on the only CSL ever tried with RNNs, namely, $a^n b^n c^n$. Traditional RNNs fail to generalize well on this simple CSL: Chalup and Blair (1999) 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 (2000) trained a sequential cascaded network with BPTT; for a training set with $n \in \{1, \dots, 10\}$, the best networks generalized to $n \in \{1, \dots, 12\}$ in 8% of the trials.

4.1 Experiment 1: online prediction

In this experiment we use LSTM with forget gates to predict subsequent symbols of a continual symbolic input stream (not segmented a priori into subsequences with clearly defined ends) with

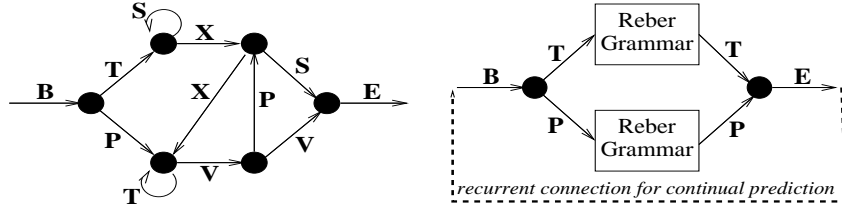


Figure 3: Transition diagrams for standard (left) and embedded (right) Reber automata. The dashed line indicates the continual variant.

long-term dependencies. The focus is on true *online* processing.

Gers et al. (2000) studied a similar problem; the difference between our approach and their related set-up is that they aborted the current input stream as soon as the network made an error, then reset the network and continued with a new input stream. In this way, the Gers et al. approach was like batch learning. On the other hand, weight updates were executed after each symbol and, as a result, their attempt can be considered as half-way between online and offline (batch) learning. Here we apply the same LSTM architecture to the same kind of sequences, but with a *pure* online approach: there is only one single input stream; learning continues even when the network makes mistakes; and training and testing are not divided into separate phases.

4.1.1 Method

We use LSTM with forget gates to predict subsequent symbols in sequences generated by the continual embedded Reber automaton (Smith and Zipser, 1989) shown in Figure 3. Due to existence of long-term dependencies, this task is suitably difficult to show the power of both LSTM and DEKF. The learning process is completely online.

We briefly review now how LSTM (in general, any neural network) can be used to predict subsequent symbols in a sequence. Consider an alphabet $\Sigma = \sigma_1, \dots, \sigma_{|\Sigma|}$ and a temporal sequence $r(1), \dots, r(t), \dots$. The number of neurons in the input and output layers equals the size of the alphabet, $|\Sigma|$. The symbols in Σ , when considered as inputs or targets, are coded by means of *local coding*, that is, σ_i is coded with a unary vector where only the i -th component is different from zero.

At time step t the symbol $r(t)$ from the sequence is coded in the input vector $\mathbf{u}(t)$ by means of local coding and presented to the network. Then the forward pass takes place, taking into account previous events represented by activations of the network’s recurrent hidden units. Finally the output vector $\mathbf{y}(t)$ is obtained and normalized such that all its components add up to one. Now $y_i(t)$ is interpreted as the probability of the next symbol being σ_i . Finally the next observed symbol $r(t+1)$ is locally encoded as vector $\mathbf{d}(t)$ and is used as the target for updating the weights in the subsequent backward pass.

4.1.2 Network topology and parameters

Following Gers et al. (2000), the LSTM network has 4 memory blocks with 2 cells each. The size of the alphabet of the automaton is $|\Sigma| = 7$, so we consider an LSTM network with 7 neurons in the input and output layers. Bias weights to input and output gates are initialized blockwise: -0.5 for the first block, -1 for the second, -1.5 for the third, and so on. Forget gate biases are initialized with symmetric values: 0.5 for the first block, 1 for the second, and so on. The rest of the weights are randomly taken from a uniform distribution in $[-0.2, 0.2]$.

The squashing function g is set to $g(x) = \tanh(x)$ with range $(-1, 1)$, $h(x)$ is set to the identity function, and the squashing function of the gates is set to the logistic sigmoid function $f(x) = (1 + e^{-x})^{-1}$ with range $(0, 1)$.

For the gradient descent training algorithm we set the learning rate to 0.5, without using an additional momentum term. In case of DEKF, the values for the free parameters of the algorithm suggested by Haykin (1999, p. 771) turned out to be adequate for this task as well: the covariance matrix of the measurement noise is annealed from 100 to 3; the covariance matrix of artificial process noise is annealed from 10^{-2} to 10^{-6} ; the elements of the initial error covariance matrix are set to 100.

4.1.3 Training and testing

We count the number of symbols needed by LSTM to attain error-free predictions for at least 1000 subsequent symbols; here “error-free” means that the symbol corresponding to the winner neuron in the network output is one of the possible transition symbols, given the current state of the Reber automaton.

Gers et al. (2000) considered longer error-free sequences, but learning was not truly online, and the networks were tested with frozen weights. Therefore, although the criterion for sustainable prediction was stringent, the learning was easier in principle. On the other hand, when working online, the recurring presence of particular subsequences usually makes the network forget past history and trust more recent observations instead. This is what one would expect from an online model, which is supposed to deal correctly with non-stationary environments.

After an initial training period, LSTM usually makes only few mistakes and tends to keep making correct predictions. To obtain a tolerant measure of prediction quality we measure the time at which the N -th error takes place after the first 1000 subsequent error-free predictions: here we consider two possible values for N , namely, 1 and 10.

4.1.4 Results

LSTM with gradient descent results. Table 1 shows the results for 9 different sequences with 9 independently initialized LSTM networks trained by the original gradient descent training algorithm. In one case (row 5) no correct prediction sequences (for 1000 symbols in a row) are found before the 1000000-th sequence symbol; this is indicated in the table by 1000000^+ .

It should be noted that the average number of symbols required for learning to predict accurately in real-time (thousands of symbols) is much smaller than the number of symbols required in Gers et al.’s (Gers et al., 1999) offline set-up (millions of symbols). This deserves more study.

LSTM with DEKF results. Performance is better using the DEKF training algorithm. The time required to achieve 1000 error-free predictions in a row is generally lower than with the original training algorithm, indicating faster convergence (compare Table 2). However the number of symbols processed before the 10-th error is also lower, indicating faster performance degradation. The DEKF seems to increase online learning speed while at the same time reducing the long-term memory capabilities of LSTM. There are three remarkable cases (rows 2, 6 and 9 in Table 2), however, where a very long subsequence (hundreds of thousands of symbols) is necessary for the 10-th error to appear.

Traditional RNNs results. Experiments with traditional RTRL-trained (Williams and Zipser, 1989) RNNs (such as the simple recurrent net (Elman, 1990) or the recurrent error

Table 1: Time steps required by online LSTM (trained with the original learning algorithm) to achieve 1000 subsequent correct predictions (“sustainable prediction”). Also shown are the number of steps before a single error and before 10 errors. Network 5 failed to make a sustainable prediction before time step 1000000.

Net	Sustainable prediction	Next error	Next 10 errors
1	39229	143563	178229
2	102812	111442	144846
3	53730	104163	141801
4	54565	58936	75666
5	1000000 ⁺	–	–
6	111483	113715	136038
7	197748	199445	235387
8	54629	123565	123595
9	85707	86742	92312

Table 2: Time steps required by online LSTM (trained by DEKF) to achieve 1000 subsequent correct predictions (“sustainable prediction”). Also shown are the number of steps before a single error and before 10 errors. Network 8 failed to make a sustainable prediction before time step 1000000. Row 6 shows a particularly good result: only 3 errors occur before the 1000000-th sequence symbol.

Net	Sustainable prediction	Next error	Next 10 errors
1	29304	30347	30953
2	19758	25488	322980
3	20487	22235	24106
4	26175	27542	33253
5	18015	19365	22241
6	16667	29826	1000000 ⁺
7	23277	24796	26664
8	1000000 ⁺	–	–
9	29742	31535	594117

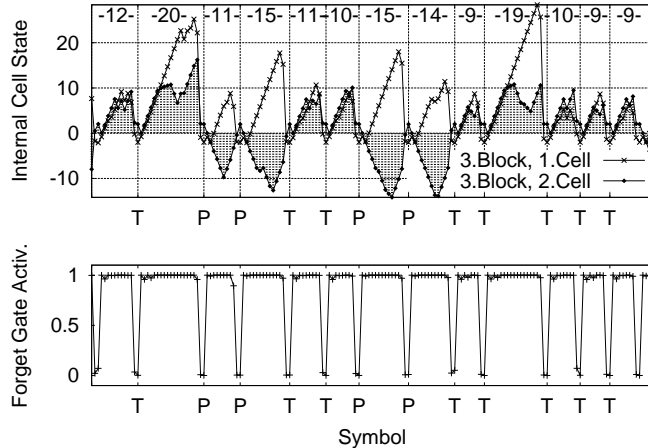


Figure 4: Top: internal states s_c of the two cells of the block memorizing long-time information during sequence prediction in an LSTM network trained on the embedded Reber grammar task. The figure shows 170 successive symbols taken from an error-free sequence. Long-time information is indicated by vertical lines labeled by the symbols (P or T) to be stored until the sequence gets the *left* side of the automaton again. Bottom: simultaneous forget gate activations for the same memory block.

propagation network (Robinson and Fallside, 1991)) demonstrated that they are unable to obtain sustainable error-free predictions for 1000 subsequent symbols, even after extremely long training times. Even as few as 100 subsequent correct predictions were extremely rare. The DEKF applied to these architectures, however, did allow for sustainable error-free predictions. But it always required many more than 100000 symbols (far more than LSTM).

4.1.5 Analysis.

Study of the evolution of gate and state activations revealed that online LSTM learns a behavior similar to the one observed in previous experiments which were not online (Gers et al., 2000, Sect. 4.4); that is, one memory block specializes in bridging long-time information (see Figure 4), while the others track short-term changes in the input only. Common to all memory blocks is that they learn to reset themselves in appropriate ways, by making the forget gate activation go to zero. This behaviour was common to both training algorithms.

4.2 Experiment 2: context sensitive language learning

We use LSTM with forget gates and the recently introduced peephole connections to learn and generalize over the CSL $a^n b^n c^n$.

4.2.1 Method

The network sequentially observes exemplary symbol strings of the 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 must be predicted, and none of the others. Thus, e.g., for $n = 3$ the inputs and targets are:

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

Every input sequence begins with the start symbol S . The empty string, consisting of ST only, is considered part of the 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 generate 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 m -dimensional vectors, where m is equal to the number of symbols of the given language $|\Sigma|$ plus one for either the start symbol in the input or the *end of string* symbol in the output (m input units, m 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 .

4.2.2 Network topology and parameters

As stated before, we are using LSTM with forget gates and the recently introduced peephole connections which link directly the CECs and the corresponding gates.

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. 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 (precise initialization is not critical here). All other weights are initialized randomly in the range $[-0.1, 0.1]$. The squashing functions g and h are 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).

4.2.3 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 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 $n \in \{1, \dots, N\}$, and (b) with $n \in \{N - 1, N\}$. For large values of N , case (b) is much harder because there is no support from short (and easier to learn) strings. In this paper we focus on $n \in \{1, \dots, 10\}$ and $n \in \{20, 21\}$. We test all sets with $n \in \{L, \dots, M\}$, $L \in \{1, \dots, N - 1\}$.

Weight updating with gradient descent. Weight changes are made after each sequence. We apply either a constant learning rate or the momentum algorithm (Plaut et al., 1986) with momentum parameter 0.99. At most 10^7 training sequences are presented; we test with $M \in \{N, \dots, 500\}$ (sequences of length ≤ 1500).

Weight updating with DEKF The *online* nature of the DEKF-LSTM algorithm forces weights to be updated after each symbol presentation. The parameters of the algorithm are

Table 3: 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^{-a}) with and without momentum (momentum parameter 0.99). Shown (from left to right, for each set with constant learning rate and with momentum) are the average number of training sequences and the percentage of correct solutions once the training set was learned.

a	(1,..., 10)				(20, 21)			
	Constant		Momentum		Constant		Momentum	
	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	68	100	-	0	1170	30	-	0
4	351	100	20	90	7450	30	-	0
5	3562	100	45	90	1205	20	127	20
6	-	0	329	100	-	0	1506	20
7	-	0	3036	100	-	0	1366	10

set as follows: 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 large 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 number of training sequences presented is 10^2 ; we test with $M \in \{N, \dots, 10000\}$ (sequences of length ≤ 30000).

4.2.4 LSTM with gradient descent results

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

LSTM worked well for a wide range of learning rates (about three orders of magnitude) as can be seen in Table 3. Use of the momentum algorithm (Plaut et al., 1986) 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 3).

4.2.5 LSTM with DEKF results

The DEKF-LSTM combination significantly improves the results of the gradient descent algorithm. Very small training sets with $n \in \{1, \dots, 10\}$ are sufficient for perfect generalization up to values of $n \in \{1, \dots, 2000\}$ and more: one of the experiments with $\delta = 10^2$ gave a generalization set with $n \in \{1, \dots, 10000\}$. 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 the strings, say, $a^{8888} b^{8888} c^{8888}$ and $a^{8888} b^{8888} c^{8889}$.

With training set $n \in \{1, \dots, 10\}$ and $\delta = 10$ the average generalization set was $n \in \{1, \dots, 434\}$ (the best generalization was $n \in \{1, \dots, 2743\}$), whereas with the original training algorithm it

Table 4: 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, $\delta = 10^{-b}$. 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.

b	(1,..., 10)		(20, 21)	
	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

was $n \in \{1, \dots, 28\}$. 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 4 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 3). 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's computational complexity per time step and weight is much larger than that of gradient descent. 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 with gradient descent. This relative CPU time is shown for DEKF in parentheses in Table 4 and can be compared directly to *number of training sequence* values in Table 3.

A comparison of gradient descent and DEKF using this relative measure reveals that the additional complexity of LSTM with DEKF 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 gradient descent performance (see Table 3 for figures).

A lesser problem of DEKF is its occasional instability. Learning usually takes place in the beginning of the training phase or never at all. All failures in Table 4 are due to this.

4.2.6 Analysis

In general, LSTM solves the $a^n b^n c^n$ problem by using a combination of two counters, instantiated separately in the two memory blocks. An example in Figure 5 shows counter s_{c_2} increasing when it encounters an a symbol and then decreasing when it encounters a b symbol. A c symbol causes the accompanying input gate y_{in_2} to close and the accompanying forget gate y_{φ_2} to reset the cell state thus emptying the counter. Counter s_{c_1} (in a different memory block) does the

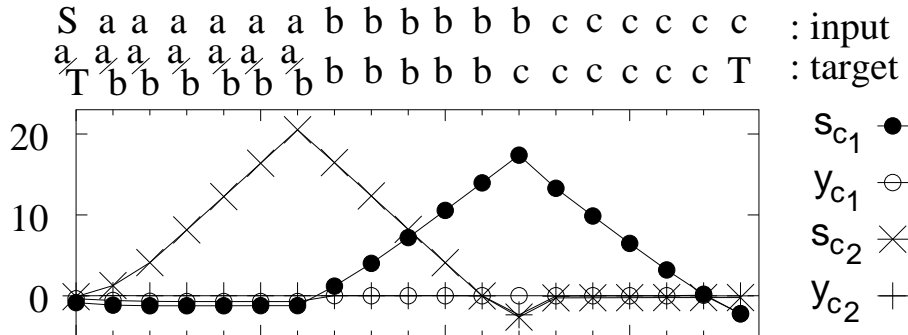


Figure 5: Test run with network solution for $a^n b^n c^n$ with $n = 6$, showing cell state s_c and cell output y_c .

same for b , c , and a , respectively. When counting up, the step size is smaller due to the closed input gate, which is triggered by s_c via the peephole connection. This results in *overshooting* the initial value of s_c which in turn triggers the highly nonlinear opening of the output gate, leading to the prediction of the sequence termination. In short, one memory block solves $a^n b^n$ while another solves $b^n c^n$. All of this works in extremely precise and robust fashion in both training algorithms.

5 Concluding remarks

LSTM combined with DEKF improves upon the original gradient descent learning algorithm making LSTM achieve even faster convergence and much better performance.

In the case of symbolic online learning the DEKF-based approach reduces significantly the number of training steps necessary for error-free prediction. However, it forgets more easily.

LSTM is the first RNN to generalize well on nonregular 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 the standard LSTM algorithm. 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, in a very stable and robust manner.

The combination DEKF-LSTM represents a general advance. The update complexity per training example, however, is worse than gradient descent. But the method is still local in time.

References

- Bakker, B., 2001. Reinforcement learning with Long Short-Term Memory. In: Dietterich, T. G., Becker, S., Ghahramani, Z. (Eds.), *Advances in neural information processing systems*, 14. MIT Press, Cambridge, MA, to appear.
- Boden, M., Wiles, J., 2000. Context-free and context-sensitive dynamics in recurrent neural networks. *Connection Science* 12 (3).
- Chalup, S., Blair, A., 1999. Hill climbing in recurrent neural networks for learning the $a^n b^n c^n$ language. In: *Proceedings of the 6th Conference on Neural Information Processing*. pp. 508–513.
- Elman, J. L., 1990. Finding structure in time. *Cognitive Science* 14, 179–211.
- Feldkamp, L. A., Puskorius, G. V., 1994. Training controllers for robustness: multi-stream DEKF. In: *IEEE International Conference on Neural Networks*. pp. 2377–2382.

- Gers, F. A., Schmidhuber, J., 2001. LSTM recurrent networks learn simple context free and context sensitive languages. *IEEE Transactions on Neural Networks* Accepted.
- Gers, F. A., Schmidhuber, J., Cummins, F., 1999. Learning to forget: continual prediction with LSTM. In: *Proc. Int. Conf. on Artificial Neural Networks*. pp. 850–855.
- Gers, F. A., Schmidhuber, J., Cummins, F., 2000. Learning to forget: continual prediction with LSTM. *Neural Computation* 12 (10), 2451–2471.
- Haykin, S., 1999. *Neural networks: a comprehensive foundation*, 2nd Edition. Prentice-Hall, New Jersey.
- Haykin, S. (Ed.), 2001. *Kalman filtering and neural networks*. Wiley.
- Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., 2001a. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: Kremer, S. C., Kolen, J. F. (Eds.), *A field guide to dynamical recurrent neural networks*. IEEE Press.
- Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. *Neural Computation* 9 (8), 1735–1780.
- Hochreiter, S., Younger, A. S., Conwell, P. R., 2001b. Learning to learn using gradient descent. In: *Lecture Notes on Comp. Sci. 2130, Proc. Intl. Conf. on Artificial Neural Networks*. Springer, Berlin, Heidelberg, pp. 87–94.
- 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., Hinton, G. E., 1986. Experiments on learning back propagation. Tech. Rep. CMU-CS-86-126, Carnegie-Mellon University, Pittsburgh, PA.
- Puskorius, G. V., Feldkamp, L. A., 1994. Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. *IEEE Transactions on Neural Networks* 5 (2), 279–297.
- Robinson, A. J., Fallside, F., 1991. A recurrent error propagation speech recognition system. *Computer Speech and Language* 5, 259–274.
- Rodriguez, P., Wiles, J., 1998. Recurrent neural networks can learn to implement symbol-sensitive counting. In: *Advances in Neural Information Processing Systems*, 10. The MIT Press, pp. 87–93.
- Rodriguez, P., Wiles, J., Elman, J., 1999. A recurrent neural network that learns to count. *Connection Science* 11 (1), 5–40.
- Schmidhuber, J., 1989. A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science* 1 (4), 403–412.
- Schmidhuber, J., 1992. A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation* 4 (2), 243–248.
- Smith, A. W., Zipser, D., 1989. Learning sequential structures with the real-time recurrent learning algorithm. *International Journal of Neural Systems* 1 (2), 125–131.
- Sun, G. Z., Giles, C. L., Chen, H. H., Lee, Y. C., 1993. The neural network pushdown automaton: model, stack and learning simulations. Tech. Rep. CS-TR-3118, University of Maryland, College Park.
- Wiles, J., Elman, J., 1995. Learning to count without a counter: a case study of dynamics and activation landscapes in recurrent networks. In: *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*. MIT Press, Cambridge, MA, pp. 482–487.
- Williams, R. J., 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., Zipser, D., 1989. A learning algorithm for continually training recurrent neural networks. *Neural Computation* 1, 270–280.

Williams, R. J., Zipser, D., 1992. Gradient-based learning algorithms for recurrent networks and their computational complexity. In: Chauvin, Y., Rumelhart, D. E. (Eds.), *Back-propagation: theory, architectures and applications*. Hillsdale, NJ, Erlbaum.