Neural Processing of Complex Continual Input Streams

Felix A. Gers Jürgen Schmidhuber
felix@idsia.ch juergen@idsia.ch
IDSIA, Corso Elvezia 36
6900 Lugano, Switzerland
www.idsia.ch

Abstract

Long Short-Term Memory (LSTM) can learn algorithms for temporal pattern processing not learnable by alternative recurrent neural networks (RNNs) or other methods such as Hidden Markov Models (HMMs) and symbolic grammar learning (SGL). Here we present tasks involving arithmetic operations on continual input streams that even LSTM cannot solve. But an LSTM variant based on "forget gates," a recent extension, has superior arithmetic capabilities and does solve the tasks.

1 Introduction

In principle, RNNs are suited for tasks no other current sequence learning method can solve. Traditional discrete SGL algorithms may faster learn grammatical structure of discrete, noise-free event sequences, but cannot deal with noise or with sequences of real-valued inputs. HMMs are well-suited for noisy inputs but also limited to discrete state spaces. Genetic Programming etc. in principle could search in general algorithm spaces but is slow due to the absence of gradient information providing a search direction. Typical RNNs, on the other hand, perform gradient descent in a very general space of potentially noise-resistant algorithms using distributed, continuous-valued internal states to map real-valued input sequences to real-valued output sequences.

Traditional RNNs (see survey: (Pearlmutter, 1995)), however, suffer from exponential decay of gradient information. In presence of long time lags between relevant input and target events they fail to learn anything within reasonable time. LSTM (Hochreiter and Schmidhuber, 1997) overcomes this problem by enforcing non-decaying error flow "back into time." Thus LSTM rather quickly solves many tasks traditional RNNs cannot solve.

Recently, however, we identified a weakness of LSTM in dealing with continual input streams that are not *a priori* segmented into separate training sequences, such that it is not clear when to reset the network's internal state. We introduced "forget gates" as a remedy (Gers et al., 1999).

Here we focus on tasks involving arithmetic operations on input streams that so far have been addressed only in non-continual settings (Tsung and Cottrell, 1989; Hochreiter and Schmidhuber, 1997). There is a level of arithmetic complexity where even standard LSTM fails in continual settings. We show that LSTM with "forget gates" exhibits superior arithmetic capabilities and does solve the tasks in an elegant way.

2 Standard LSTM

The basic unit in the hidden layer of an LSTM network is the *memory block*, which contains one or more *memory cells* and a pair of adaptive, multiplicative gating units which gate input and output to 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 we call the cell *state*. In principle the CECs solve the error decay problem (Hochreiter and Schmidhuber, 1997): in the absence of new input or error signals to the cell, the CEC's local error back flow remains constant. The CEC is protected from both forward flowing activation and backward flowing error by the input and output gates respectively. When gates are closed (activation around zero), irrelevant inputs and noise do not enter the cell, and the cell state does not perturb the remainder of the network. Fig. 1 shows a memory block with a single cell. The cell state, s_c, is updated

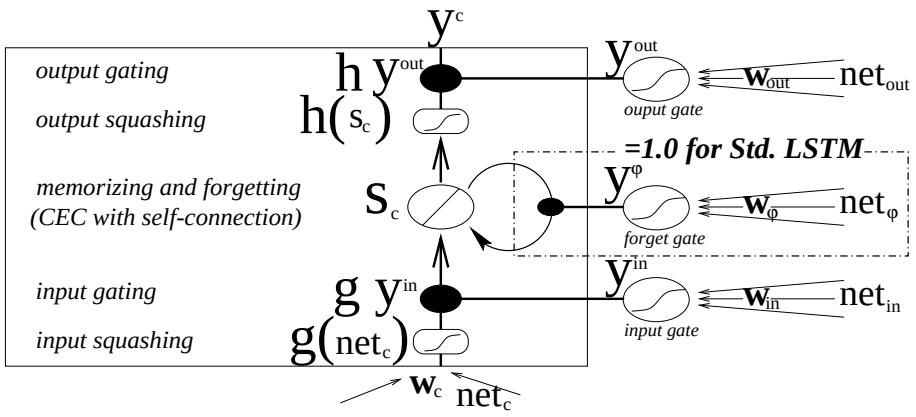

Figure 1: The LSTM cell has a linear unit with a recurrent self-connection. Input and output gates regulate read and write access to the cell whose state is denoted by s_c. The function g squashes the cell's input; h squashes the cell's output (see text for details).

based on its current state and three sources of input: net_c is input to the cell itself; net_{in} and net_{out} are inputs to the input and output gates.

We consider discrete time steps $t = 1, 2, \ldots$. A single step involves the update of all units (forward pass) and the computation of error signals for all weights (backward pass). Input gate activation y^{in} and output gate activation y^{out} are computed as follows:

$$y^{in_j}(t) \;=\; f_{in_j}(\sum_m w_{in_j m}\; y^m(t\text{–}1)) \;, \quad y^{out_j}(t) = f_{out_j}(\sum_m w_{out_j m}\; y^m(t\text{–}1)) \;\; . \tag{1}$$

Throughout this paper j indexes memory blocks; v indexes memory cells in block j, such that c_j^v denotes the v-th cell of the j-th memory block (with S_j cells); 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. For gates, f is a logistic sigmoid with range $[0, 1]$. Net input to the cell itself is squashed by g. The internal state of memory cell $s_c(t)$ is calculated by adding the squashed, gated input to the state at the previous time step $s_c(t\text{–}1)$ $(t > 0)$:

$$s_{c_j^v}(t) = s_{c_j^v}(t\text{–}1) + y^{in_j}(t)\; g(net_{c_j^v}(t)) \;, \tag{2}$$

with $s_{c_j^v}(0) = 0$. The cell output y^c is calculated by squashing the internal state s_c via the output squashing function h, and then multiplying (gating) it by the output gate activation y^{out}:

$$y^{c_j^v}(t) = y^{out_j}(t)\; h(s_{c_j^v}(t)) \;\; . \tag{3}$$

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:

$$y^k(t) = f_k(\sum_m w_{km}\; y^m(t\text{–}1)) \;\; , \tag{4}$$

where f_k is a squashing function and m ranges over all units feeding the output units (typically all cells in the hidden layer, but not the memory block gates).

Learning. See (Hochreiter and Schmidhuber, 1997) for details of standard LSTM's backward pass. Essentially, as in truncated BPTT, errors arriving at net inputs of memory blocks and their gates do not get propagated back further in time, although they *do* serve to change the incoming weights. When an error signal arrives at a memory cell output, it gets scaled by the output gate and the output nonlinearity h; it then enters the memory cell's linear CEC, where it can flow back indefinitely without change (thus LSTM can bridge arbitrary time lags between input events and target signals). Only when the error escapes from the memory cell through an open input gate and the additional input nonlinearity g, does it get scaled once more and then serves to change incoming weights before being truncated. The update complexity is $O(1)$ per weight and time step. Hence LSTM is also computationally cheaper than BPTT and RTRL.

2.1 Limits of standard LSTM.

Memorizing for too long. LSTM allows information to be stored across arbitrary time lags, and error signals to be carried far back in time. This potential strength, however, can contribute to a weakness in some situations: the cell states s_c often tend to grow linearly during the presentation of a time series (the nonlinear aspects of sequence processing are left to the squashing functions and the highly nonlinear gates). In presence of a continual input stream the cell states may grow in unbounded fashion, causing saturation of the output squashing function, h. Saturation will (a) make h's derivative vanish, thus blocking incoming errors, and (b) make the cell output equal the output gate activation, that is, the entire memory cell will degenerate into an ordinary BPTT unit. This problem did not arise in the experiments reported in (Hochreiter and Schmidhuber, 1997) because cell states were explicitly reset to zero before the start of each new sequence. Our solution to the above problem is to use adaptive "forget gates" (Gers et al., 1999), which learn to reset memory blocks once their contents are out of date and hence useless.

Arithmetic capabilities. Due to its architecture LSTM is well suited for tasks involving addition, subtraction and integration (Hochreiter and Schmidhuber, 1997). Such operations are essential for many real-world tasks. But another essential arithmetic operation, namely *multiplication*, does pose problems. Forget gates, however, originally introduced to release irrelevant memory contents, greatly improve LSTM's performance on tasks involving multiplication, as will be seen below.

3 Forget Gates

Standard LSTM's constant CEC weight 1.0 (Fig. 1) is replaced by the multiplicative forget gate activation y^{φ}. The forget gate activation y^{φ} is calculated like the activations of the other gates and squashed using a logistic sigmoid with range $[0, 1]$:

$$y^{\varphi_j}(t) \quad = \quad f_{\varphi_j}(\sum_m w_{\varphi_j m}\ y^m(t\text{–}1)) \ . \tag{5}$$

y^{φ} functions as the weight of the self recurrent connection of the internal state s_c in equation (2). The revised update equation for s_c in the extended LSTM algorithm is (for $t > 0$):

$$s_{c_j^v}(t) \quad = \quad y^{\varphi_j}(t)\ s_{c_j^v}(t\text{–}1) + \ y^{in_j}(t)\ g(net_{c_j^v}(t)) \ , \tag{6}$$

with $s_{c_j^v}(0) = 0$. Bias weights for LSTM gates are initialized with negative values for input and output gates (see (Hochreiter and Schmidhuber, 1997) for details), positive values for forget gates. This implies that in the beginning of the training phase the forget gate activation will be almost 1.0, and the entire cell will behave like a standard LSTM cell. It will not explicitly forget anything until it has learned to forget.

3.1 Backward Pass

LSTM's backward pass (see (Gers et al., 1999; Hochreiter and Schmidhuber, 1997) for details) is an efficient fusion of slightly modified, truncated BPTT, and a customized version of RTRL. The squared error objective function based on targets t^k is:

$$E(t) = \frac{1}{2}\sum_k e_k(t)^2 \quad ; \quad e_k(t) := t^k(t) - y^k(t) \ .$$

We minimize E via gradient descent weight changes $\Delta w_{lm} = \alpha\ \delta_l(t)\ y^m(t\text{–}1)$, using learning rate α. Here $\delta_k(t) = f_k'(net_k(t))\ e_k(t)$ for output units k and

$$\delta_{out_j}(t) \quad = \quad f'_{out_j}(net_{out_j}(t)) \ \left(\sum_{v=1}^{S_j} h(s_{c_j^v}(t)) \sum_k w_{kc_j^v}\ \delta_k(t)\right) \tag{7}$$

for the output gate of the j-th memory block.

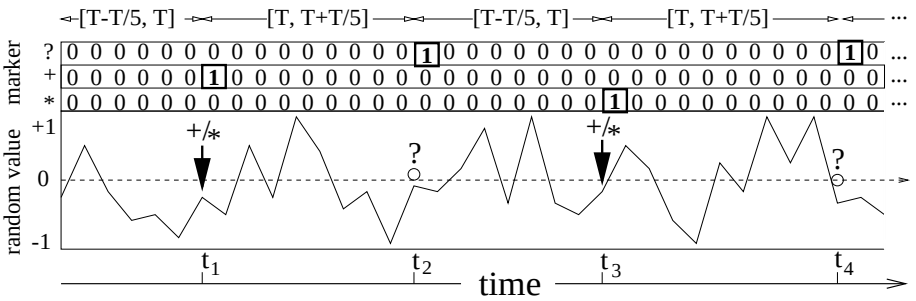

Figure 2: Illustration of the continual addition (and multiplication) tasks.

For weights to cell, input gate and forget gate we adopt an RTRL-oriented perspective: We define the internal state error $e_{s_{c_j^v}}$ and the partial $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ of $s_{c_j^v}$ with respect to weights w_{lm} feeding the cell c_j^v $(l = c_j^v)$ or the block's input gate $(l = in)$ or the block's forget gate $(l = \varphi)$. For each cell, $e_{s_{c_j^v}}$ is given by:

$$e_{s_{c_j^v}}(t) = y^{out_j}(t)\ h'(s_{c_j^v}(t)) \left(\sum_k w_{kc_j^v}\ \delta_k(t) \right) \tag{8}$$

The partials $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ for $l \in \{\varphi, in, c_j^v\}$ are zero for $t = 0$. For $t > 0$ we get:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}}\ y^{\varphi_j}(t) + g'(net_{c_j^v}(t))\ y^{in_j}(t)\ y^m(t-1)\ , \tag{9}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}}\ y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ f'_{in_j}(net_{in_j}(t))\ y^m(t-1)\ , \tag{10}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}}\ y^{\varphi_j}(t) + s_{c_j^v}(t-1)\ f'_{\varphi_j}(net_{\varphi_j}(t))\ y^m(t-1)\ . \tag{11}$$

Updates of weights to the cell $\Delta w_{c_j^v m}$ only depend on the partials of this cell's own state, whereas updating the weights of the input gate $(l = in)$ and of the forget gate $(l = \varphi)$ requires to sum over the contributions of all cells in the block:

$$\Delta w_{c_j^v m}(t) = \alpha\ e_{s_{c_j^v}}(t)\ \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \quad , \quad \Delta w_{lm}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t)\ \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \quad . \tag{12}$$

The storage complexity for the backward pass does not depend on the length of the input sequence. The update complexity is $O(1)$ per weight and time step. Compare to BPTT (peak storage complexity $O(S)$, where S is sequence length) and RTRL (update complexity $O(W^2)$ per time step, where W is the number weights) (Williams and Zipser, 1989; Robinson and Fallside, 1987). Like standard LSTM but unlike BPTT and RTRL, extended LSTM is *local in space and time*, despite its superior long time lag capabilities.

4 Experiments

Many typical real world sequence processing tasks involve continual input streams, distributed input representations, continuous-valued targets and inputs and internal states, and long time lags between relevant events. So we designed several artificial nonlinear tasks that combine these factors. Note that the combination excludes all other sequence learning methods (SGL, HMMs, BPTT, RTRL, etc.) from being applicable at all!

General set-up. We feed the net continual streams of 4-dimensional input vectors generated in an online fashion. We define $t_0 = 0$ (stream start) and $t_n = t_{n-1} + T + (-1)^n \cdot V$ for $n = 1, 2, \ldots$, where

$V \in \{0, 1, \ldots, \frac{T}{5}\}$ is chosen randomly, and integer T is the minimal time lag. The first component of each input vector is a random number from the interval $[-1, +1]$. The second and third serve as "markers": they are always 0.0 except at times t_{2n-1} when either the second component is 1.0 with probability p, or the third is 1.0 with probability $1-p$. The fourth component is always 0 except at times t_{2n} when targets are given and its activation is 1.0. The target at t_0 is 0. If the 2nd component was active at t_{2n-1} then the target at t_{2n} is the sum of the previous target at t_{2n-2} and the "marked" first input component at t_{2n-1}. Otherwise it is the product of these two values.

Hence non-initial targets depend on events that happened at least $2 \cdot T$ steps ago. Note that occurrences of "value markers" and targets oscillate. See Figure 2 for an illustration of the task.

All streams are stopped once the absolute output error exceeds 0.04. Test streams are almost unlimited (max. length = 1000 target occurrences), but training streams end after at most 10 target occurrences. Learning and testing alternate: after each training stream we freeze the weights and feed 100 test streams. Our performance measure is the average test stream size.

Task 1: Continual addition. $p = 1.0$ (no multiplication). $T = 20$. Task 1 essentially requires to keep adding (possibly negative) values to the already existing internal state.

Task 2: Continual addition and multiplication. $p = 0.5$, $T = 20$. If the 3rd input component is active at t_{2n-1} and the 1st is negative then the latter will get replaced by its absolute value.

Task 3: Gliding addition. Like Task 1, but targets at times t_{2n+2} equal the sum of the two most recent marked values at times t_{2n+1} and t_{2n-1} (the first target at t_2 equals the first value at t_1). $T = 10$. Task 3 is harder than task 1 because it requires selective partial resets of the current internal state.

4.1 Network Topology and Parameters

The 4 input units are fully connected to a hidden layer consisting of 3 memory blocks with 1 cell each (roughly: less blocks decreased performance for standard LSTM and more blocks did not improve performance significantly). The cell outputs are fully connected to the cell inputs, to all gates, and to the output unit. All gates and output units are biased. Bias weights to in- and output gates are initialized block-wise: -1.0 for the first block, -2.0 for the second, and so forth. (This is a standard initialization procedure: blocks with higher bias tend to get released later during the learning phase - the precise initialization is not at all important though.) Forget gates are initialized with symmetric positive values: $+1.0$ for the first block, $+2.0$ for the second, and so forth. The squashing functions g,h and f_k are the identity function.

4.2 Results

See Table 1. Test stream sizes are measured by number of target presentations before first failure. A stream size below 3 counts as an unsuccessful trial. We report the best test performance during a training phase involving $3 \cdot 10^6$ training streams, averaged over 10 independent networks.

Task 1. Both standard LSTM and LSTM with forget gates learn the task. Worse performance of LSTM with forget gates is caused by slower convergence, because the net has to learn to remember everything and not to forget.

Task 2. LSTM with forget gates solves the problem even when addition and multiplication are combined, whereas standard LSTM's solutions are not sufficiently accurate. This shows that forget gates add algorithmic functionality to memory blocks besides releasing resources during runtime (their original purpose which is not essential here).

Task 3. Standard LSTM cannot solve the problem at all, whereas LSTM with forget gates does find good and even "perfect" solutions. Why? The forget gates learn to prevent LSTM's uncontrolled internal state growth (see section 2.1), by resetting states once stored information becomes obsolete.

Other Experiments. To compare standard LSTM and extended LSTM we ran numerous additional experiments. In particular, we created continual variants of hard tasks in (Hochreiter and Schmidhuber, 1997). In general, standard LSTM (and alternative recurrent nets) failed, while LSTM with forget gates succeeded. On the other hand we have not found a task yet that standard LSTM can solve but LSTM with forget gates cannot (Gers et al., 1999).

The results indicate that forget gates are a mandatory patch for LSTM fed with continual input streams, where obsolete memories need to be discarded at some point (see "Task 3: Gliding addition"). Experiment

Table 1: Average test stream size (percentage of successful trials given in parenthesis). In Task 3 one network with forget gates exceeded the limit of 1000 target occurrences.

Algorithm	Task 1	Task 2	Task 3
Standard LSTM	73 (100%)	- (0%)	- (0%)
LSTM + Forget Gates	42 (100%)	40 (60%)	241 (50%)

2 shows that forget gates also greatly facilitate operations involving multiplication.

5 Conclusion

LSTM, a method local in space an time, can extract sequence-processing algorithms from training sequences whose regularities remain opaque to all other approaches we are aware of. While previous work focused on training sequences with well-defined beginnings and ends, however, typical real-world input streams are not *a priori* segmented into training subsequences indicating network resets. Therefore RNNs should be able to *learn* appropriate self-resets. This is also desirable for tasks with hierarchical but *a priori* unknown decompositions. For instance, re-occurring subtasks should be solved by the same network module, which should be reset once the subtask is solved. Forget gates naturally permit LSTM to learn local self-resets of memory contents that have become irrelevant. They also substantially improve LSTM's performance on tasks involving arithmetic operations.

The tasks reported above are artificial but do exhibit aspects of realistic tasks, such as continual input streams, distributed input representations, continuous-valued targets and inputs and internal states, long time lags. How about real world tasks? In ongoing work we are applying extended LSTM to raw continual speech data. We have started to successfully extract additional relevant prosodic information neglected by traditional, phoneme-based HMM approaches (Cummins et al., 1999).

Acknowledgments

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

References

Cummins, F., Gers, F., and Schmidhuber, J. (1999). Language identification from prosody without explicit features. In *Proceedings of EUROSPEECH'99*, volume 1, pages 371–374.

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. Extended version submitted to Neural Computation.

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

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

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.

Tsung, F. S. and Cottrell, G. W. (1989). A sequential adder using recurrent networks. In *Proceedings of the First International Joint Conference on Neural Networks, Washington, DC*, San Diego. IEEE, IEEE TAB Neural Network Committee.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent net works. *Neural Computation*, 1(2):270–280.