FORSCHUNGSBERICHTE KÜNSTLICHE INTELLIGENZ

Long Short Term Memory

Sepp Hochreiter, Jürgen Schmidhuber

Report FKI-207-95

August 1995

T U M

TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik (H2), D-80290 München, Germany

ISSN 0941-6358

Forschungsberichte Künstliche Intelligenz

ISSN 0941-6358

Institut für Informatik
Technische Universität München

Die Forschungsberichte Künstliche Intelligenz enthalten vornehmlich Vorab-Veröffentlichungen, spezialisierte Einzelergebnisse und ergänzende Materialien, die seit 1988 in der KI / Kognitionsgruppe am Lehrstuhl Prof. Brauer bzw. 1988-1993 in der KI / Intellektik Gruppe am Lehrstuhl Prof. Jessen entstanden. Im Interesse einer späteren Veröffentlichung wird gebeten, die Forschungsberichte nicht zu vervielfältigen. Alle Rechte und die Verantwortung für den Inhalt des Berichts liegen bei den Autoren, die für kritische Hinweise dankbar sind.

Eine Zusammenstellung aller derzeit lieferbaren FKI-Berichte und einzelne Exemplare aus dieser Reihe können Sie bei folgender Adresse anfordern oder über ftp beziehen:

"FKI"
Institut für Informatik (H2)
Technische Universität München
D-80290 München
Germany

Phone: +49 - 89 - 2105 2406
Telex: tumue d 05-22854
Fax: +49 - 89 - 2105 - 8207
e-mail: fki@informatik.tu-muenchen.de

The "Forschungsberichte Künstliche Intelligenz" series includes primarily preliminary publications, specialized partial results, and supplementary material, written by the members of the AI / Cognition Group at the chair of Prof. Brauer (since 1988) as well as the "Intellektik" Group at the chair of Prof. Jessen (1988-1993). In the interest of a subsequent final publication these reports should not be copied. All rights and the responsibility for the contents of the report are with the authors, who would appreciate critical comments.

You can obtain a list of all available FKI-reports as well as specific papers by writing to the adress below or via ftp:

FTP:
machine: flop.informatik.tu-muenchen.de
or 131.159.8.35
login: anonymous
directory: pub/fki

LONG SHORT TERM MEMORY
Technical Report FKI-207-95

Sepp Hochreiter
Fakultät für Informatik
Technische Universität München
80290 München, Germany
hochreit@informatik.tu-muenchen.de

Jürgen Schmidhuber
IDSIA
Corso Elvezia 36
6900 Lugano, Switzerland
juergen@idsia.ch

August 21, 1995

Abstract

"Recurrent backprop" for learning to store information over extended time periods takes too long. The main reason is insufficient, decaying error back flow. We describe a novel, efficient "Long Short Term Memory" (LSTM) that overcomes this and related problems. Unlike previous approaches, LSTM can learn to bridge *arbitrary* time lags by enforcing *constant* error flow. Using gradient descent, LSTM explicitly learns when to store information and when to access it. In experimental comparisons with "Real-Time Recurrent Learning", "Recurrent Cascade-Correlation", "Elman nets", and "Neural Sequence Chunking", LSTM leads to many more successful runs, and learns much faster. Unlike its competitors, LSTM can solve tasks involving minimal time lags of more than 1000 time steps, even in noisy environments.

1 INTRODUCTION

In principle, recurrent nets can use their feedback connections to store representations of recent input events in "short term memory". This is potentially significant for many applications, including speech processing, non-Markovian control, and music composition (e.g. Mozer, 1992).

However, previous algorithms for learning *what* to put in short term memory take too much time or don't work at all, especially when there are *long* time lags between inputs and corresponding teacher signals.

For instance, with conventional "backprop through time" (BPTT, e.g. Williams and Zipser, 1992) or RTRL (e.g. Robinson and Fallside, 1987), error signals "flowing backwards in time" tend to either (1) blow up or (2) vanish: the temporal evolution of the backpropagated error exponentially depends e.g. on weights of self-connections (leading from some unit to itself). See Hochreiter (1991) for a detailed analysis not limited to self-connections. Case (1) leads to oscillating weights. In case (2), learning to bridge long time lags takes a prohibitive amount of time, or does not work at all. The approaches in Elman (1988), Fahlman (1991), Williams (1989), and Schmidhuber (1992a) suffer from the same problems. Other methods that seem practicable for short time gaps only are Time-Delay Neural Networks (Lang et al., 1990) and Plate's method (Plate, 1993) (which updates unit activations based on a weighted sum of old activations, see also de Vries and Principe, 1991).

To deal with long time lags, Mozer (1992) uses time constants influencing the activation changes. However, for long time gaps the time constants need external fine tuning (Mozer, 1992). Sun et al.'s alternative approach (1993) updates the activation of a recurrent unit by adding the old activation and the (scaled) current net input. The net input, however, tends to perturb the stored information, which again makes long term storage impracticable. Schmidhuber's chunker systems *do* have a capability to bridge very long time lags, but only if the input sequence exhibits

locally predictable regularities (see Schmidhuber, 1992b; Schmidhuber et al., 1993; and Mozer, 1992).

"Long Short Term Memory" (LSTM), the new approach presented in this paper, overcomes the problems above. Unlike chunking systems, even in noisy, highly unpredictable environments, LSTM can learn loss-free information storage spanning arbitrary time periods. A major LSTM feature is that it enforces **constant, non-exploding, non-vanishing** error flow. Unlike previous approaches, ours quickly learns to distinguish between two or more occurrences of an element in an input sequence. Constant error backprop also makes the method fast (see experiments).

Outline. For didactic purposes, the next section will introduce a naive approach to constant error backprop, and highlight its problems concerning information storage and retrieval. These problems will be solved by the LSTM architecture to be described in section 3. Section 4 will present experimental comparisons with competing methods. LSTM outperforms them.

2 CONSTANT ERROR BACKPROP

Conventional BPTT (e.g. Williams and Zipser, 1992). Output unit k's target at time t is $d_k(t)$. Using mean squared error, k's error signal is $\vartheta_k(t) = f'_k(net_k(t))(d_k(t) - y^k(t))$, where $y^i(t) = f_i(net_i(t))$ is the activation of a non-input unit i with activation function f_i, w_{ij} is the weight on the connection from unit j to i and $net_j(t) = \sum_i w_{ji} y^i(t-1)$ is unit j's current net input. Some non-output unit j's backpropagated error signal is $\vartheta_j(t) = f'_j(net_j(t)) \sum_i w_{ij} \vartheta_i(t+1)$. The corresponding contribution to w_{ij}'s total weight update is $\alpha \vartheta_i(t) y^j(t-1)$, where α is the learning rate.

Constant error flow: naive approach. Consider a single unit j with a single connection to itself. According to the rules above, at time t, j's local error back flow is $\vartheta_j(t) = f'_j(net_j(t))\vartheta_j(t+1)w_{jj}$. To avoid exploding or vanishing error signals (see introduction), we wish to enforce *constant* error flow through unit j. Towards this end, we require $f'_j(net_j(t))w_{jj} = 1$. Integrating this differential equation, we obtain $f_j(net_j(t)) = \frac{net_j(t)}{w_{jj}}$ for arbitrary $net_j(t)$. This means: f_j has to be linear, and unit j's activation has to remain constant: $y_j(t+1) = f_j(net_j(t+1)) = f_j(w_{jj} y^j(t)) = y^j(t)$. In the experiments, we will use $f_j \equiv id$, and $w_{jj} = 1$.

Of course, in reality unit j will not only be connected to itself but also to other units. This invokes two obvious problems (also inherent in all other previous approaches):

1. Input weight conflict: for simplicity, let's focus on a single additional input weight w_{jl}. Assume that the total error can be reduced by switching on unit j in response to a certain input, and keeping it active for a long time (until it helps to compute a desired output). Provided i is non-zero, w_{ji} will often receive conflicting weight update signals during this time (recall: j is linear): these signals will attempt to make w_{ji} participate (1) in storing the input (by switching on j) *and* (2) in protecting the input (by preventing j from being switched off by insignificant later inputs). This conflict makes learning difficult, and calls for a more context-sensitive mechanism for controlling "write operations" through input weights.

2. Output weight conflict: for simplicity, let's focus on a single additional output weight w_{kj}. As long as unit j is non-zero, w_{kj} will attract conflicting weight update signals generated during sequence processing: these signals will attempt to make w_{kj} participate in (1) accessing the information stored in j *and* – at different times – in (2) protecting unit k from being perturbed by j. Again, this conflict makes learning difficult, and calls for a more context-sensitive mechanism for controlling "read operations" through output weights.

Due to the problems above, the naive approach does not work well. The next section shows how to do it right.

3 LONG SHORT TERM MEMORY

Memory cells and gate units (see also Hochreiter, 1991). To obtain constant error flow without the disadvantages of the naive approach, we extend the self-connected, linear unit j from section 2 by introducing additional features. The resulting, more complex unit is called a *memory cell* and is denoted c_j. In addition to net_{c_j}, c_j gets input from a special unit out_j called an "output gate", and from another special unit in_j called an "input gate". in_j's activation at time t is denoted by $y^{in_j}(t)$. out_j's activation at time t is denoted by $y^{out_j}(t)$. in_j, out_j are ordinary hidden units.

At time t, c_j's output $y^{c_j}(t)$ is computed in a sigma-pi-like fashion:

$$y^{c_j}(t) = y^{out_j}(t)h(s_{c_j}(t)),$$

where the "internal state" $s_{c_j}(t)$ is

$$s_{c_j}(0) = 0, s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t)g\left(net_{c_j}(t)\right) \text{ for } t > 0.$$

The differentiable function g scales net_{c_j}. The differentiable function h scales memory cell outputs computed from the internal state s_{c_j}.

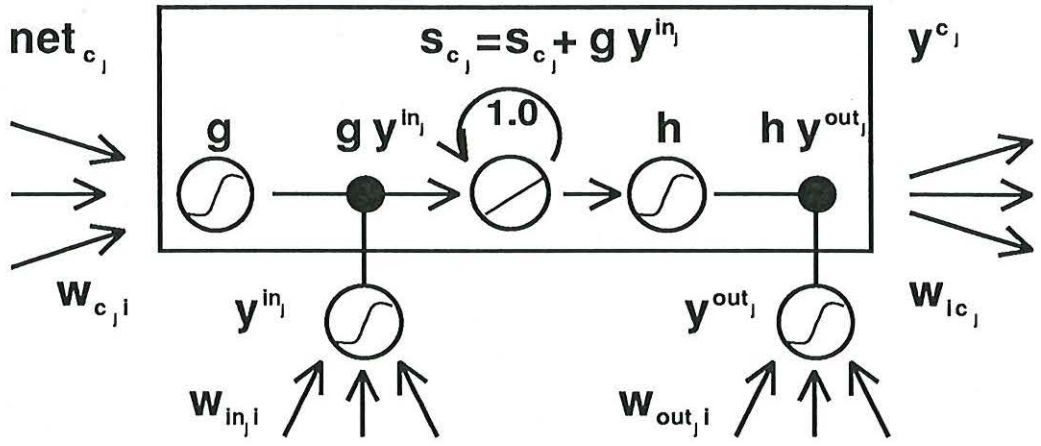

Figure 1: *Architecture of memory cell c_j (the box) and corresponding gate units in_j, out_j. See text for details.*

Why gate units? To avoid the input weight conflict problem (see section 2), in_j controls the error flow to memory cell c_j's input connections $w_{c_j i}$. The net can use in_j to decide when to keep or override information in memory cell c_j (see figure 1). To circumvent c_j's output weight conflict problem (see section 2), j_{out} controls the error flow from unit j's output connections. The net can use out_j to decide when to access memory cell c_j and when to prevent other units from being perturbed by c_j (see figure 1).

Network topology. We use a network with an input layer, a hidden layer, and an output layer. The fully self-connected hidden layer consists of memory cells and corresponding gate units. All units (except gate units) in all layers are connected to all units in higher layers.

Learning / Computational complexity. We use a variant of RTRL (e.g. Robinson and Fallside, 1987) which properly takes into account the altered (sigma-pi-like) dynamics caused by input and output gates. However, to ensure constant error backprop, like with truncated BPTT (e.g. Williams and Zipser, 1992), errors arriving at "memory cell net inputs" (for cell c_j, this includes net_{c_j}, net_{in_j}, net_{out_j}) do not get propagated back further in time (although they *do* serve to change the incoming weights). Only within memory cells, errors are propagated back through previous internal states s_{c_j}. This enforces constant error flow within memory cells. Thus, like with Mozer's focused recurrent backprop algorithm (Mozer, 1989), only the derivatives $\frac{\partial s_{c_j}}{\partial w_{il}}$ need

to be stored and updated. *This in turn implies $O(n)$ update complexity per time step, where n is the number of weights: the algorithm is very efficient.*

Abuse problem and solutions. In the beginning of the learning phase, error reduction may be possible even without storing information over time. Then the net will tend to abuse memory cells, e.g. as bias cells. The difficulty is: it may take a long time to release abused memory cells and make them available for further learning. A similar "abuse problem" appears if two memory cells store the same (redundant) information. We investigate three solutions to the abuse problem: *(1) Sequential network construction* (e.g. Fahlman, 1991): a memory cell and the corresponding gate units are added to the network whenever the error stops decreasing (see experiment 1 in section 4). *(2) Output gate bias:* each output gate gets a negative initial bias, to push memory cell activations towards zero. Memory cells with more negative bias automatically get "allocated" later (see experiment 2 in section 4). *(3) Output gate competition:* lateral inhibition ensures that no two ouput gates can be active simultaneously (see experiment 2 in section 4).

Memory cell blocks. N memory cells sharing the same input gate and the same output gate form a new structure called a "memory cell block of size N". Memory cell blocks can store more information than a single memory cell. In experiment 2 (section 4), we will use a memory cell block of size 2.

4 EXPERIMENTS

4.1 EXPERIMENT 1: SEQUENCE PREDICTION

Task. There are $p+1$ possible input symbols denoted by $a_1, ..., a_{p-1}, a_p = x, a_{p+1} = y$. a_i is "locally" represented by the $p+1$-dimensional vector whose ith component is 1 (all other components are 0). A net with $p+1$ input units and $p+1$ output units sequentially observes input symbol sequences, one at a time, permanently trying to predict the next symbol. To emphasize the "long time lag problem", we use a training set consisting of only two very similar sequences: $(y, a_1, a_2, \ldots, a_{p-1}, y)$ and $(x, a_1, a_2, \ldots, a_{p-1}, x)$. To predict the final element, the net has to learn to store a representation of the first element for p time steps.

We compare "Real-Time Recurrent Learning" (RTRL, e.g. Robinson and Fallside, 1987), the sometimes very successful neural sequence chunker (CH, Schmidhuber, 1992b), and our new method (LSTM). In all cases, weights are initialized in [-0.2,0.2]. Training is stopped after 5 million sequence presentations. Success is defined as "maximal absolute output error of all units always below 0.25".

Architectures. RTRL: one self-recurrent hidden unit, $p+1$ non-recurrent output units. Each layer has connections from all layers below. All units sigmoid in [0,1].

CH: both nets like with RTRL above, but one has an additional output for predicting the hidden unit of the other one (see Schmidhuber, 1992b for details).

LSTM: like with RTRL, but the hidden unit is replaced by a memory cell and an input gate (no output gate required). g is sigmoid in [0,1] and $h \equiv id$. Memory cell and input gate are added once the error has stopped decreasing (see abuse problem: solution (1) in section 3).

Results. Using RTRL and a 4 time step delay ($p = 4$), $\frac{7}{9}$ of all trials were successful. *No trial was successful with $p = 10$.* With *long* time lags, only the neural sequence chunker and the new approach achieved successful trials. With $p = 100$, the sequence chunker solved the task in only $\frac{1}{3}$ of all trials. *LSTM, however, always learned to solve the task.* Comparing successful trials only, *LSTM learned much faster.* See table 1 for details.

EXPERIMENT 1b: no local regularities. With the task above, CH sometimes learns to correctly predict the final element, *but only because of predictable local regularities in the input stream that allow for compressing the sequence.* In an additional, more difficult task (involving many more different possible sequences), we remove compressibility by replacing the deterministic subsequence $(a_1, a_2, \ldots, a_{p-1})$ by a *random* subsequence (of length $p-1$) over the alphabet $a_1, a_2, \ldots, a_{p-1}$. As expected, *the chunker failed to solve this task. Our new approach, however, was always successful.* On average, success was achieved after 5,680 sequence presentations (mean

Method	Delay p	Learning rate	% Successful trials	Success after
RTRL	4	1.0	78	1,043,000
RTRL	4	4.0	56	892,000
RTRL	4	10.0	22	254,000
RTRL	10	1.0-10.0	0	> 5,000,000
RTRL	100	1.0-10.0	0	> 5,000,000
CH	100	1.0	33	32,400
LSTM	100	1.0	100	5,040

Table 1: *Percentage of successful trials and number of training sequences until success, for "Real-Time Recurrent Learning" (RTRL), neural sequence chunking (CH), and new method (LSTM). Table entries refer to the mean of 18 trials. With 100 time step delays, only CH and LSTM achieve successful trials.* **Even when we ignore the unsuccessful trials of the other approaches: LSTM learns much faster.**

of 18 trials). This illustrates: the new approach does not depend on sequence regularities.

EXPERIMENT 1c: very long time lags — no local regularities. There are $p+4$ possible input symbols denoted $a_1, ..., a_{p-1}, a_p, a_{p+1} = e, a_{p+2} = b, a_{p+3} = x, a_{p+4} = y$. $a_1, ..., a_p$ are also called *"distractor symbols"*. Again, a_i is locally represented by the $p+4$-dimensional vector whose ith component is 1 (all other components are 0). A net with $p+4$ input units and 2 output units sequentially observes input symbol sequences, one at a time. The training set is the union of two very similar subsets of sequences: $\{(b, y, a_{i_1}, a_{i_2}, \ldots, a_{i_{q+k}}, e, y) \mid 1 \leq i_1, i_2, \ldots, i_{q+k} \leq q\}$ and $\{(b, x, a_{i_1}, a_{i_2}, \ldots, a_{i_{q+k}}, e, x) \mid 1 \leq i_1, i_2, \ldots, i_{q+k} \leq q\}$. To pick a training sequence, we first select some non-negative integer k with probability $P(k) = \frac{1}{10}(\frac{9}{10})^k$. Once k is selected, a training sequence is generated according to a uniform distribution on the possible sequences with length $q+k+4$. The minimal sequence length is $q+4$. The expected sequence length is $q + 14 = 4 + \sum_{k=0}^{\infty} P(k)(q+k)$. The expected number of occurences of element $a_i, 1 \leq i \leq p$, in a sequence is $\frac{q+10}{p} \approx \frac{q}{p}$. The goal is to predict the last symbol, which always occurs after the "trigger symbol" e. To predict the final element, the net has to learn to store a representation of the second element for at least $q+1$ time steps (until it sees the trigger symbol e). Success is defined as "prediction error (for final sequence element) of both output units always below 0.2".

Architecture / Learning. Weights are initialized in [-0.2,0.2]. To avoid too much learning time variance due to different weight initializations, the hidden layer has two memory cells (although one would be sufficient). There are no other hidden units. No unit is biased. h is sigmoid in $[-1, 1]$, and g is sigmoid in $[-2, 2]$. This allows for pushing absolute memory cell outputs towards 1.0. Error signals occur only for predictions of the final sequence element. The learning rate is always 0.01. Note that the *minimal* time lag is always $q+1$ — the net never sees short training sequences facilitating the classification of long test sequences.

Results. 20 trials were made for all tested pairs (p, q). Table 2 lists the mean of the number of training sequences required by LSTM to achieve success (of course, RTRL and the other competitors have no chance of solving tasks with time lags involving 1000 time steps).

Scaling. Table 2 shows: if we let the number of input symbols (and weights) increase in proportion to the time lag, learning time increases very slowly. This is a another remarkable property of LSTM not shared by any other architecture we are aware of. Indeed, architectures like RTRL are far from scaling reasonably — instead, they appear to scale exponentially, and appear quite useless when the time lags exceed as few as 10 time steps.

Distractor influence. In Table 2, the column headed by $\frac{q}{p}$ reflects the expected frequency of distractor symbols. Increasing this frequency decreases learning speed. This effect is due to weight oscillations caused by frequently observed input symbols.

q (time lag -1)	p (# random inputs)	$\frac{q}{p}$	Success after
50	50	1	30,000
100	100	1	31,000
200	200	1	33,000
500	500	1	38,000
1,000	1,000	1	49,000
1,000	500	2	49,000
1,000	200	5	75,000
1,000	100	10	135,000
1,000	50	20	203,000

Table 2: *LSTM with very long minimal time lags $q+1$. p is the number of available distractor symbols. $\frac{q}{p}$ is the expected number of occurences of a given distractor symbol in a sequence. The last column lists the number of training sequences required by LSTM (of course, RTRL and the other competitors have no chance of solving tasks with time lags involving 1000 time steps).* **If we let the number of distractor symbols (and weights) increase in proportion to the time lag, learning time increases very slowly.** *The lower block illustrates the expected performance slow-down due to increased frequency of distractor symbols.*

4.2 EXPERIMENT 2: EMBEDDED REBER GRAMMAR

Task. Symbol strings are produced by the "embedded Reber grammar", which is often used as a benchmark for recurrent networks, e.g. Smith and Zipser (1989), Cleeremans et al. (1989) and Fahlman (1991). Again, the task is to read strings, one symbol at a time, and to permanently predict the next symbol.

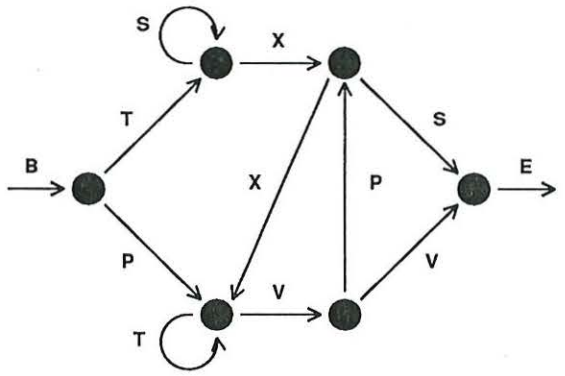

Figure 2: *Transition diagram for the Reber grammar.*

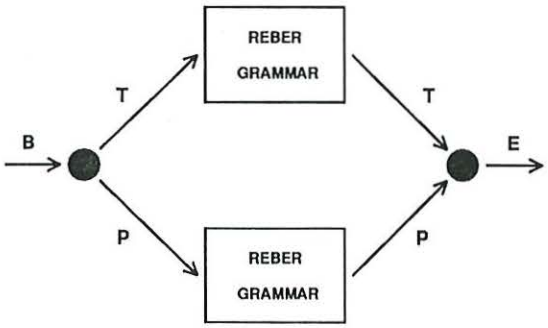

Figure 3: *Transition diagram for the embedded Reber grammar. Each box represents a copy of a Reber grammar (see figure 2).*

Starting at the leftmost node of the directed graph in figure 3, legal strings are generated sequentially (beginning with the empty string) by following edges (and appending the corresponding symbols to the current string), until the rightmost node is reached. Edges are chosen randomly if there is a choice (probability: 0.5). This task is not trivial: to predict the symbol before the last one, the net has to remember the second symbol.

Comparison. We compare RTRL (results taken from Smith and Zipser (1989), where only the few successful trials are listed), the "Elman net" (ELM) (results taken from Cleeremans et al., 1989), Fahlman's "Recurrent Cascade-Correlation" (RCC) (results taken from Fahlman, 1991), and our new method (LSTM).

Training / Testing. We use local input/output representation as in section 4.1 (7 input

method	hidden units	learning rate	% of success	success after
RTRL	3	0.05	"some fraction"	173,000
RTRL	12	0.1	"some fraction"	25,000
ELM	15		0	>200,000
RCC	7-9		50	182,000
LSTM	3 blocks, size 2	0.5	100	8,440

Table 3: *Embedded Reber grammar: percentage of successful trials and number of sequence presentations until success for RTRL (results taken from Smith and Zipser, 1989), "Elman net" (results taken from Cleeremans et al., 1989), "Recurrent Cascade-Correlation" (results taken from Fahlman, 1991) and our new approach (LSTM).* **Only LSTM always learns to solve the task. Even when we ignore the unsuccessful trials of the other approaches: LSTM learns much faster** *(the number of required training examples varies from 3,800 to 24,100).*

units, 7 output units). Following Fahlman, we use 256 training strings and 256 separate test strings. After string presentation, all activations are reinitialized with zeros. A trial is considered successful if all string symbols of all sequences in both test set and training set are predicted correctly: if the output unit(s) corresponding to the possible next symbol(s) is(are) always the most active ones.

Architectures. Architectures for RTRL, ELM, RCC are reported in the references listed above. For LSTM, we use 3 memory cell blocks. Each block has 2 memory cells. All activation functions are sigmoid in $[0, 1]$, except for h, which is sigmoid in $[-1, 1]$, and g, which is sigmoid in $[-2, 2]$. This allows for pushing the absolute memory cell outputs towards 1.0. All weights are initialized in $[-0.2, 0.2]$. The initial output gate biases are $-1, -2, -3$ (see abuse problem solution (2) of section 3). The learning rate is 0.5.

Results. We use 3 different, randomly generated pairs of training sets and test sets. With each such pair, 10 trials with different weight initializations are made. See table 3 for results (mean of 30 trials). Unlike the other methods, *LSTM always learns to solve the task. Even when we ignore the unsuccessful trials of the other approaches, LSTM learns much faster.*

With additional output gate competition (solution (3) of the abuse problem of section 3), learning was even faster.

5 CONCLUSION

Long Short Term Memory represents a significant improvement over previous neural algorithms for dealing with arbitrary, unknown, temporal delays between input and target events.

References

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite-state automata and simple recurrent networks. *Neural Computation*, 1:372–381.

de Vries, B. and Principe, J. C. (1991). A theory for neural networks with time delays. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 162–168. San Mateo, CA: Morgan Kaufmann.

Elman, J. L. (1988). Finding structure in time. Technical Report CRL 8801, Center for Research in Language, University of California, San Diego.

Fahlman, S. E. (1991). The recurrent cascade-correlation learning algorithm. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 190–196. San Mateo, CA: Morgan Kaufmann.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

Lang, K., Waibel, A., and Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3:23–43.

Mozer, M. C. (1989). A focused back-propagation algorithm for temporal sequence recognition. *Complex Systems*, 3:349–381.

Mozer, M. C. (1992). Induction of multiscale temporal structure. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 275–282. San Mateo, CA: Morgan Kaufmann.

Plate, T. A. (1993). Holographic recurrent networks. In S. J. Hanson, J. D. C. and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 34–41. San Mateo, CA: Morgan Kaufmann.

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.

Schmidhuber, J. H. (1992a). A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2):243–248.

Schmidhuber, J. H. (1992b). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242.

Schmidhuber, J. H., Mozer, M. C., and Prelinger, D. (1993). Continuous history compression. In Hüning, H., Neuhauser, S., Raus, M., and Ritschel, W., editors, *Proc. of Intl. Workshop on Neural Networks, RWTH Aachen*, pages 87–95. Augustinus.

Smith, A. W. and Zipser, D. (1989). Learning sequential structures with the real-time recurrent learning algorithm. *International Journal of Neural Systems*, 1(2):125–131.

Sun, G., Chen, H., and Lee, Y. (1993). Time warping invariant neural networks. In S. J. Hanson, J. D. C. and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 180–187. San Mateo, CA: Morgan Kaufmann.

Williams, R. J. (1989). Complexity of exact gradient computation algorithms for recurrent neural networks. Technical Report Technical Report NU-CCS-89-27, Boston: Northeastern University, College of Computer Science.

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

FKI-207-95 Sepp Hochreiter, Jürgen Schmidhuber: Long Short Term Memory

ISSN 0941-6358