Long Short-Term Memory in Recurrent Neural Networks

THÈSE N° 2366 (2001)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

FELIX GERS

Diplom in Physik, Universität Hannover, Deutschland
de nationalité allemand

soumise à l'approbation du jury:

Prof. R. Hersch, président
Prof. Wulfram Gerstner, directeur de thèse
Dr. habil. Jürgen Schmidhuber, corapporteur
Prof. Paolo Frasconi, corapporteur
Dr. MER Martin Rajman, corapporteur

Lausanne, EPFL
2001

Long Short-Term Memory in Recurrent Neural Networks

Contents

1 Introduction **5**
- 1.1 Recurrent Neural Networks (RNNs) . . . 5
- 1.2 General considerations. . . . 6
  - 1.2.1 Problem: Exponential decay of gradient information. . . . 7
  - 1.2.2 Solution: Constant error carousels. . . . 7
- 1.3 Previous and Related Work . . . 7
  - 1.3.1 RNNs . . . 7
  - 1.3.2 RNNs versus Other Sequence Processing Approaches . . . 8
- 1.4 Outline . . . 9

2 Traditional LSTM **11**
- 2.1 Forward Pass . . . 12
- 2.2 Learning . . . 13
- 2.3 Tasks Solved with Traditional LSTM . . . 14

3 Learning to Forget: Continual Prediction with LSTM **15**
- 3.1 Introduction . . . 15
  - 3.1.1 Limits of traditional LSTM . . . 15
- 3.2 Solution: Forget Gates . . . 16
  - 3.2.1 Forward Pass of Extended LSTM with Forget Gates . . . 16
  - 3.2.2 Backward Pass of Extended LSTM with Forget Gates . . . 17
  - 3.2.3 Complexity . . . 20
- 3.3 Experiments . . . 21
  - 3.3.1 Continual Embedded Reber Grammar Problem . . . 21
  - 3.3.2 Network Topology and Parameters . . . 23
  - 3.3.3 CERG Results . . . 24
  - 3.3.4 Analysis of the CERG Results . . . 25
  - 3.3.5 Continual Noisy Temporal Order Problem . . . 25
- 3.4 Conclusion . . . 28

4 Arithmetic Operations on Continual Input Streams **29**
- 4.1 Introduction . . . 29
- 4.2 Experiments . . . 29
  - 4.2.1 Network Topology and Parameters . . . 30
  - 4.2.2 Results . . . 30
- 4.3 Conclusion . . . 31

5 Learning Precise Timing with Peephole LSTM **33**
5.1 Introduction . . . 33
5.2 Extending LSTM with "Peephole Connections" . . . 34
5.3 Forward Pass . . . 35
5.4 Gradient-Based Backward Pass . . . 36
5.5 Experiments . . . 37
5.5.1 Network Topology and Experimental Parameters . . . 38
5.5.2 Measuring Spike Delays (MSD) . . . 39
5.5.3 Generating Timed Spikes (GTS) . . . 43
5.5.4 Periodic Function Generation (PFG) . . . 44
5.5.5 General Observation: Network initialization . . . 51
5.6 Conclusion . . . 51

6 Simple Context Free and Context Sensitive Languages **53**
6.1 Introduction . . . 53
6.2 Experiments . . . 54
6.2.1 Training and Testing . . . 55
6.2.2 Network Topology and Experimental Parameters . . . 55
6.2.3 Previous results . . . 56
6.2.4 LSTM Results . . . 57
6.2.5 Analysis . . . 58
6.3 Conclusion . . . 62

7 Time Series Predictable Through Time-Window Approaches **63**
7.1 Introduction . . . 63
7.2 Experimental Setup . . . 64
7.2.1 Network Topology . . . 65
7.3 Mackey-Glass Chaotic Time Series . . . 65
7.3.1 Previous Work . . . 66
7.3.2 Results . . . 66
7.3.3 Analysis . . . 67
7.4 Laser Data . . . 70
7.4.1 Previous Work . . . 71
7.4.2 Results . . . 72
7.4.3 Analysis . . . 72
7.5 Conclusion . . . 73

8 Conclusion **75**
8.1 Main Contributions . . . 75
8.2 Future work and possible applications of LSTM. . . . 76

A Embedded Reber Grammar Statistics **77**

B Peephole LSTM with Forget Gates in Pseudo-code **79**

References **83**

Personal Record **90**

Acknowledgments **94**

This thesis is based on the following publications:

Chapter 2

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

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999b). Learning to forget: Continual prediction with LSTM. In *Proc. ICANN'99, Int. Conf. on Artificial Neural Networks* (Vol. 2, p. 850-855). Edinburgh, Scotland: IEE, London.

Chapter 3

Gers, F. A., & Schmidhuber, J. (2000c). Neural processing of complex continual input streams. In *Proc. IJCNN'2000, Int. Joint Conf. on Neural Networks.* Como, Italy.

Chapter 4

Gers, F. A., & Schmidhuber, J. (2000e). Recurrent nets that time and count. In *Proc. IJCNN'2000, Int. Joint Conf. on Neural Networks.* Como, Italy.

Gers, F. A., Schmidhuber, J., & Schraudolph, N. Learning precise timing with LSTM recurrent networks. (submitted to Neural Computation)

Chapter 5

Gers, F. A., & Schmidhuber, J. (2001). Long short-term memory learns simple context free and context sensitive languages. *IEEE Transactions on Neural Networks.* (accepted)

Gers, F. A., & Schmidhuber, J. Long short-term memory learns context free and context sensitive languages. In *ICANNGA 2001 Conference.* (accepted)

Chapter 6

Gers, F. A., Eck, D., & Schmidhuber, J. (2001). Applying LSTM to time series predictable through time-window approaches. In *Proc. ICANN 2001, Int. Conf. on Artificial Neural Networks.* Vienna, Austria: IEE, London. (submitted)

Other publications: (IDSIA technical reports are not listed, see www.idsia.ch.)

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999a). Continual prediction using LSTM with forget gates. In M. Marinaro & R. Tagliaferri (Eds.), *Neural Nets, WIRN Vietri-99, Proceedings 11th Workshop on Neural Nets* (p. 133-138). Vietri sul Mare, Italy: Springer Verlag, Berlin.

Gers, F. A., & Schmidhuber, J. (2000a). LSTM learns context free languages. In *Snowbird 2000 Conference.*

Cummins, F., Gers, F., & Schmidhuber, J. (1999). Language identification from prosody without explicit features. In *Proceedings of EUROSPEECH'99* (Vol. 1, pp. 371–37 4).

Abstract

For a long time, recurrent neural networks (RNNs) were thought to be theoretically fascinating. Unlike standard feed-forward networks RNNs can deal with arbitrary input sequences instead of static input data only. This combined with the ability to memorize relevant events over time makes recurrent networks in principal more powerful than standard feed-forward networks. The set of potential applications is enormous: any task that requires to learn how to use memory is a potential task for recurrent networks. Potential application areas include time series prediction, motor control in non-Markovian environments and rhythm detection (in music and speech).

Previous successes in real world applications, with recurrent networks were limited, however, due to practical problems when long time lags between relevant events make learning difficult. For these applications conventional gradient-based recurrent network algorithms for learning to store information over *extended* time intervals take too long. The main reason for this failure is the rapid decay of back-propagated error. The "Long Short Term Memory" (LSTM) algorithm overcomes this and related problems by enforcing *constant* error flow. Using gradient descent, LSTM explicitly learns when to store information and when to access it.

In this thesis we extend, analyze, and apply the LSTM algorithm. In particular, we identify two weaknesses of LSTM, offer solutions and modify the algorithm accordingly: (1) We recognize a weakness of LSTM networks processing continual input streams that are not *a priori* segmented into subsequences with explicitly marked ends at which the network's internal state could be reset. Without resets, the state may grow indefinitely and eventually cause the network to break down. Our remedy is a novel, adaptive "forget gate" that enables an LSTM cell to learn to reset itself at appropriate times, thus releasing internal resources. (2) We identify a weakness in LSTM's connection scheme, and extend it by introducing "peephole connections" from LSTM's "Constant Error Carousel" to the multiplicative gates protecting them. These connections provide the gates with explicit information about the state to which they control access. We show that peephole connections are necessary for numerous tasks and do not significantly affect LSTM's performance on previously solved tasks.

We apply the extended LSTM with forget gates and peephole connections to tasks that no other RNN algorithm can solve (including traditional LSTM): Grammar tasks and temporal order tasks involving continual input streams, arithmetic operation on continual input streams, tasks that require precise, continual timing, periodic function generation and context free and context sensitive language tasks. Finally we establish limits of LSTM on time series prediction problems solvable by time window approaches.

Sommario

Per molto tempo le reti neuronali ricorrenti sono state considerate teoricamente affascinanti. Le reti ricorrenti possono trattare naturalmente sequenze di dati invece di poter ricevere solo dati statici in input. Possono imparare a memorizzare gli eventi importanti. Queste capacitá le rendono in linea di principio piú potenti delle reti feed-forward. La classe di potenziali applicazioni è ampia: essa contiene ogni problema che richiede l'uso di memoria interna. Alcuni esempi sono la previsione delle serie storiche (time series prediction), il controllo del moto (motor control) in ambienti non Markoviani e il riconoscimento del ritmo (per esempio nella musica o nella lingua parlata).

D'altra parte, sinora le reti ricorrenti anno avuto poco successo nell'applicazione a problemi reali caratterizzati da intervalli temporali lunghi tra eventi importanti dell'input. Gli algoritmi convenzionali di apprendimento, basati sul gradiente, hanno bisogno di troppo tempo per imparare a memorizzare delle informazioni con intervalli temporali lunghi. La ragione principale è la rapida decrescita dell'errore retropropagato (back-propagation error). Le reti di tipo long-short term memory (LSTM) offrono una soluzione a questo problema, proponendo un'architettura dove il flusso dell'errore rimane costante. Usando l'inclinazione del gradiente (gradient descent), le reti LSTM possono imparare quando un'informazione deve essere memorizzata e quando va successivamente usata.
Questa tesi analizza, estende ed applica l'algoritmo LSTM. Si identificano due difetti dell'algoritmo preesistente é si propongono due estensioni principali dell'algoritmo che risolvono i problemi riscontrati. In particolare: (1) viene identificato un difetto dell'algoritmo LSTM che accade quando l'input é contiguo, cioè non a priori suddiviso in sottosequenze con inizi e fini distinti. In questo caso, l'algoritmo non è in grado di determinare quando la rete va riportata allo stato iniziale e i valori interni possono crescere illimitatamente causando una paralisi del sistema. Il rimedio proposto si basa su una nuova unità moltiplicativa (gate unit) adattabile chiamato "forget gate". Essa permette ad una cella della rete LSTM di imparare a ritornare ad uno stato precedente in momenti opportuni, liberando così risorse interne.
(2) Si identifica un difetto nello schema delle connessioni delle reti LSTM e lo si risolve introducendo connessioni chiamate "peephole connections". Esse collegano l'unità centrale ("constant error carousel") delle celle alle unità moltiplicative che le stanno attorno. In questo modo vengono fornite alle unitá informazioni esplicite sulla condizione dell'oggetto di cui controllano l'accesso. Si mostra inoltre che le peephole connections sono necessarie per numerosi problemi e che non riducono significativamente la performance delle reti LSTM su problemi precedentemente affrontati.

La tesi applica l'algoritmo LSTM esteso con forget gates e peephole connections a problemi che nessun altro algoritmo per reti ricorrenti può risolvere (compreso le reti LSTM tradizionali): problemi di grammatica; problemi di ordinamenti temporali che coinvolgono input continui; operazioni aritmetiche su input continuo; problemi che richiedono una continua e precisa misura del tempo; la generazione di funzioni periodiche e riconoscimento di grammatiche context-free e context-sensitive. Infine si identificano dei limiti dell'algoritmo LSTM esteso relativi a problemi di previsione di serie storiche che sono risolvibili dalla classe di metodi basati su finestre temporali.

Chapter 1

Introduction

The goal of this Ph.D. thesis is to extend, analyze, and apply a recent, novel, promising gradient learning algorithm for recurrent neural networks (RNNs). The algorithm is called *"Long Short Term Memory" (LSTM)*. It was introduced by Hochreiter and Schmidhuber (1997).

1.1 Recurrent Neural Networks (RNNs)

The RNNs we consider here consist of units interacting in discrete time via directed, weighted connections with weights w_{lm} (from unit m to unit l). Every unit has an activation $y(t)$ updated at every time $t = 1, 2, \ldots$. The activations of the units feeding into other units form the state of the network. An activation y^l of unit l is updated by computing its network input sum net^l,

$$net^l(t) = \sum_m w_{lm}\ y^m(t-1)\ ,$$

and "squashing" it with a differentiable function f, according:

$$y^l(t) = f(net^l(t))\quad .$$

Input and output to the network are time-varying series of vector-patterns called sequences.

We define learning in RNNs as optimizing a differentiable objective function E, summed over all time steps of all sequences. by adapting the connection weights. E is based on supervised targets t^k, where k indexes the output units of the network with activations y^k. An example is the squared error objective function:

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

where e_k denotes the externally injected error; $E(t)$ represents the error at time t for one sequence component called pattern. For a typical data set consisting of sequences of patterns, E is the sum of $E(t)$ over all patterns of all sequences in the set.

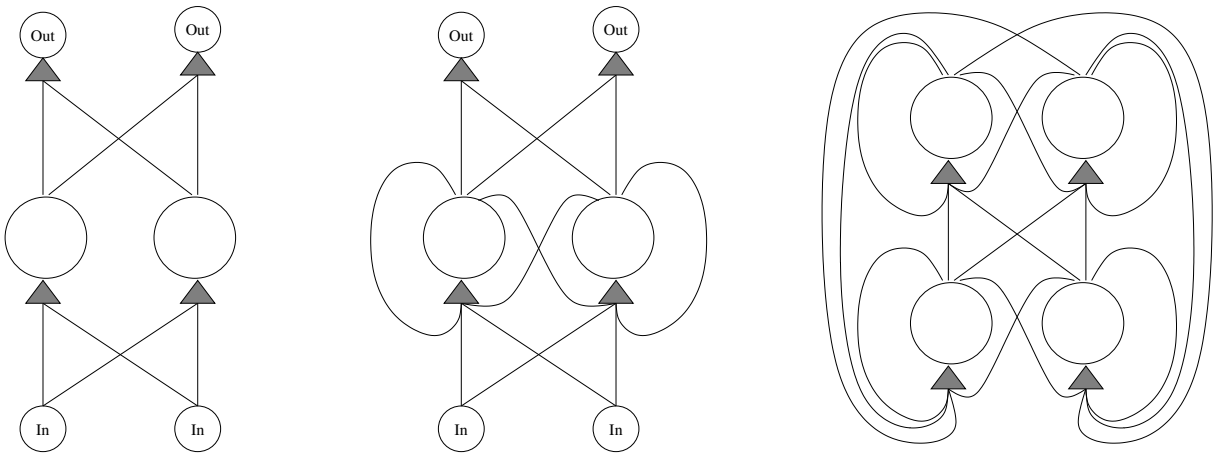

Figure 1.1: Left: Feed-forward neural network. Middle: Layered network with an input layer, a fully recurrent hidden layer and an output layer. Right: Fully connected recurrent network.

A gradient descent learning algorithm for RNNs, such as LSTM, computes the gradient of E with respect to each weight w_{lm} to determine the weight changes Δw_{lm}:

$$\Delta w_{lm}(t) \quad = \quad -\alpha \, \frac{\partial E(t)}{\partial w_{lm}} \quad ,$$

where α is called the learning rate. For an excellent introduction to gradient learning in RNNs see Williams and Zipser (1992).

The connection scheme of a network is called the network architecture or topology. Architectures without loops are called feed-forward neural networks (Figure 1.1, left). RNN topologies range from partly recurrent, to fully recurrent networks. An example of a partly recurrent network is a layered network with distinct input and output layers, where the recurrence is limited to the hidden layer(s), as shown in the middle of Figure 1.1. In fully recurrent networks each node gets input from all other nodes (Figure 1.1, right).

1.2 General considerations.

RNNs constitute a very powerful class of computational models, capable of instantiating almost arbitrary dynamics (Siegelmann & Sontag, 1991).

Usually two basic types of RNN are distinguished: autonomous RNNs with converging dynamics where the input is fixed, for example Hopfield networks (Hopfield, 1982) or Boltzmann machines (Hinton, Sejnowski, & Ackley, 1984), versus non-autonomous RNNs with time-varying inputs. The RNNs considered in this thesis are of the latter class. They can perform gradient descent in a very general space of potentially noise-resistant algorithms using distributed, continuous-valued internal states. The ability to map real-valued input sequences to real-valued output sequences, making use of their internal state to incorporate past context, makes them a remarkably general sequence processing devices (Bengio, Frasconi, Gori, & G.Soda, 1993). RNNs are especially promising for tasks that require to learn how to use memory. Potential applications are: time series prediction (e.g., of financial series), time series production (e.g., motor control in non-Markovian environments) and time series classification or labeling (e.g., rhythm detection in music and speech).

1.2.1 Problem: Exponential decay of gradient information.

The extent to which this potential can be exploited, is however limited by the effectiveness of the training procedure applied. Gradient based methods (see survey: (Pearlmutter, 1995))—"Back-Propagation Through Time" (Williams & Zipser, 1992; Werbos, 1988) or "Real-Time Recurrent Learning" (Robinson & Fallside, 1987; Williams & Zipser, 1992) and their combination (Schmidhuber, 1992a)—share an important limitation. The temporal evolution of the path integral over all error signals "flowing back in time" exponentially depends on the magnitude of the weights (Hochreiter, 1991). This implies that the back-propagated error quickly either vanishes or blows up (Hochreiter & Schmidhuber, 1997; Bengio, Simard, & Frasconi, 1994; Schmidhuber, 1992b). Hence standard RNNs fail to learn in the presence of long time lags between relevant input and target events. Tasks with time lags greater than 5–10 already become difficult for then to learn within reasonable time (Hochreiter & Schmidhuber, 1997). The vanishing error problem casts doubt on whether standard RNNs can indeed exhibit significant practical advantages over time-window-based feed-forward networks (Hochreiter, 1991; Bengio et al., 1994).

1.2.2 Solution: Constant error carousels.

The LSTM algorithm overcomes this problem by enforcing non-decaying error flow "back into time." It can learn to bridge minimal time lags in excess of 1000 discrete time steps (Hochreiter & Schmidhuber, 1997) by enforcing *constant* error flow through "constant error carrousels" (CECs) within special units, called cells. Multiplicative gate units learn to open and close access to the cells. Thus LSTM rather quickly solves many tasks traditional RNNs cannot solve. LSTM's learning algorithm is local in space and time; its computational complexity per time step and weight for standard topologies is $O(1)$.

1.3 Previous and Related Work

In this section we will give a brief overview on alternative approaches for time series processing and provide pointers to the original works. Throughout the thesis we will discuss their applicability for the various tasks we investigate.

Time window approaches. Any static pattern matching device (e.g., feed-forward network) with a fixed time window of recent inputs can serve as temporal sequence processing system. This approach has several significant drawbacks: (1) It is difficult to determine the optimal time window size, it there is any. (2) For tasks with long-term dependencies a large input window is necessary. A solution might be to use a combination of several time windows. But this is only applicable when the exact long-term dependencies of the task are known, which is usually not the case. (3) Fixed time windows are inadequate when a task has changing long-term dependencies.

An RNN approach, on the other hand, can avoid these problems, because RNNs do in principle not need access to the past. They can potentially learn to extract and represent a Markov state.

1.3.1 RNNs

Elman networks and RNNs with context units. In Elman network the content of the hidden units is copied into so called context units, which feed back into the hidden layer (Elman,

1990). (This topology is equivalent to a network with a hidden layer, where each unit feeds into every other one via time delayed connections with delay one.) Elman nets are trained by back-propagation (Rumelhart, Hinton, & Williams, 1986); thus they do not even propagate errors back though time. In alternative approaches with context units the hidden units feed (e.g., fully connected) into the context units (their number may be different from the number of the hidden units). Usually BPTT or RTRL (and their truncated versions) are used for training.

Time delay neural networks (TDNNs). Time-Delay Neural Networks (TDNNs) (Haffner & Waibel, 1992) allow access to past events via cascaded internal delay lines. The interval they can access depends on the network topology. Thus they suffer from the same problems as feed-forward networks using a time window.

Nonlinear autoregressive models with exogenous inputs (NARX) networks. NARX networks (Lin, Horne, Tiño, & Giles, 1996), allow for several distinct input time-windows (possibly of size one) with different temporal offsets. They can potentially solve tasks with stationary long time lags; it remains a problem to determine the right windows. However, when the long term dependencies are non-stationary the approach fails.

Focused back-propagation. To deal with long time lags, Mozer (1989) uses time constants which influence 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.

Continual, Hierarchical, Incremental Learning and Development (CHILD). Ring (1994) proposed the CHILD method for bridging long time lags. Whenever a unit in his network receives conflicting error signals, he adds a higher order unit influencing appropriate connections. Although his approach can sometimes be extremely fast, to bridge a time lag involving 100 steps may require the addition of 100 units. The network cannot generalize to sequences with unseen lag durations.

Chunker systems. Chunker systems (Schmidhuber, 1992b; Mozer, 1992) do have the ability to bridge arbitrary time lags, but only if the input sequence exhibits locally predictable regularities.

LSTM. LSTM does not suffer from the problems above. It seems to be the state of the art method for recurrent networks faced with realistic, long time lags between occurrences of relevant events.

1.3.2 RNNs versus Other Sequence Processing Approaches

Discrete symbolic grammar learning algorithms (SGLAs). SGLAs (Lee, 1996; Sakakibara, 1997) may faster learn grammatical structure of discrete, noise-free event sequences, but cannot deal well with noise or with sequences of real-valued inputs (Osborne & Briscoe, 1997).

Hidden Markov models (HMMs). HMMs are widely used approaches to sequence processing. They are well-suited for noisy inputs and are invariant to non-linear temporal stretching. This makes HMMs especially successful in speech recognition (they do not care for the difference between slow and fast versions of a given spoken word). But for many other tasks HMMs are less suited, because, unlike RNNs, they are limited to discrete state spaces. This makes their application to many time series task cumbersome and inefficient. For example for simples counting tasks, HMMs need as many states as the the number of symbols on the longest sequence that should be counted. Whereas with RNNs the necessary algorithm can be

instantiated with networks of 2-5 units (Kalinke & Lehmann, 1998; Rodriguez & Wiles, 1998; Gers & Schmidhuber, 2000e). Thus, in principle RNNs are applicable to tasks beyond the reach of HMMs.

Input output hidden Markov models (IOHMMs). The input-output HMM architecture (Bengio & Frasconi, 1995) combines elements of mixture-of-experts, RNNs, and hidden Markov models, and is adapted via the EM algorithm. To our knowledge, this architecture has not yet been applied to tasks comparable to the ones discussed here. But it was shown to solve simple tasks involving long time lags.

Genetic Programming and Program Search. Genetic Programming (see e.g., Dickmanns et al., 1987; Cramer, 1985; Koza, 1992) and Probabilistic Incremental Program Evolution (PIPE) (Sałustowicz & Schmidhuber, 1997) in principle could search in general algorithm spaces but are slow due to the absence of gradient information providing a search direction.

Random guessing. For some simple benchmarks weight guessing finds solutions faster than elaborate gradient algorithms (Hochreiter & Schmidhuber, 1996, 1995; Schmidhuber & Hochreiter, 1996).

1.4 Outline

Traditional LSTM. Chapter 2 describes the traditional LSTM algorithm as introduced by Hochreiter and Schmidhuber (1997).

Forget Gates. In Chapter 3 we identify 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 introduce "forget gates" as a remedy (Gers, Schmidhuber, & Cummins, 2000, 1999b).

Arithmetic operations. In Chapter 4 we present tasks involving arithmetic operations on continual input streams that traditional LSTM cannot solve. But LSTM extended with forget gates has superior arithmetic capabilities and does solve the tasks (Gers & Schmidhuber, 2000c).

Timing, extending LSTM with "peephole connections". In Chapter 5 we investigate tasks where the temporal distance between events conveys essential information (this is the case for numerous sequential tasks such as motor control and rhythm detection). First we identify a weakness in LSTM's connection scheme, regarding the wiring of the nonlinear, multiplicative gates surrounding and protecting LSTM's constant error carrousels (CEC). We extend LSTM by introducing "peephole connections" from the CECs to the gates and find that LSTM augmented by peephole connections can learn precise timing. It learned, for example, the fine distinction between sequences of spikes separated by either 50 or 49 discrete time steps, without the help of any short training exemplars (Gers & Schmidhuber, 2000e; Gers, Schmidhuber, & Schraudolph,).

Context free and context sensitive languages. Previous work by Hochreiter and Schmidhuber (1997) and the our work (see Chapter 3) showed that LSTM outperforms traditional RNNs on learning regular languages from exemplary training sequences. In Chapter 6 we demonstrate LSTM's superior performance on context free language (CFL) benchmarks for recurrent neural networks (RNNs). To the best of our knowledge, LSTM variants are also the first RNNs to learn a simple context *sensitive* language (CSL), namely $a^n b^n c^n$ (Gers & Schmidhuber, 2001,).

Time series prediction. In Chapter 7 LSTM is applied to time series prediction tasks solvable by time window approaches: the Mackey-Glass series and the Santa Fe FIR laser emission series (Set A) (Gers, Eck, & Schmidhuber, 2000, 2001).

Chapter 2

Traditional LSTM

The basic unit in the hidden layer of an LSTM network is the *memory block*; it replaces the hidden units in a "traditional" RNN (Figure 2.1). A memory block 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. Memory blocks allow cells to share the same gates (provided the task permits this), thus reducing the number of adaptive parameters. 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*. The CEC's solve the vanishing error problem: in the absence of new input or

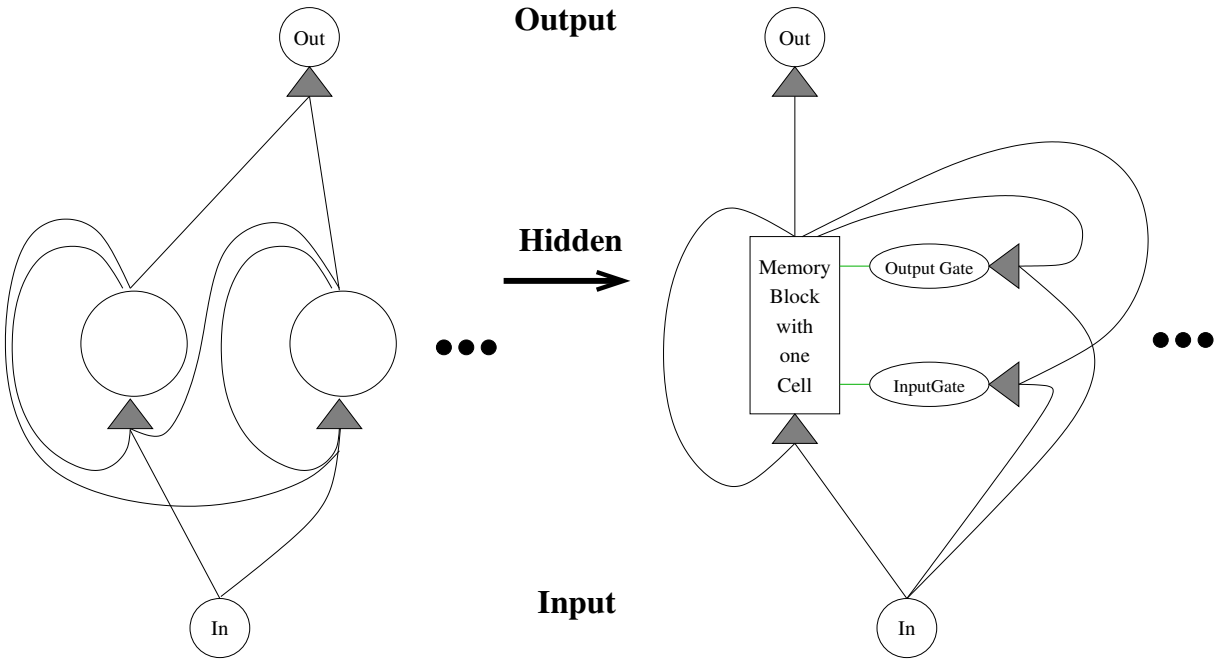

Figure 2.1: Left: RNN with one fully recurrent hidden layer. Right: LSTM network with memory blocks in the hidden layer (only one is shown).

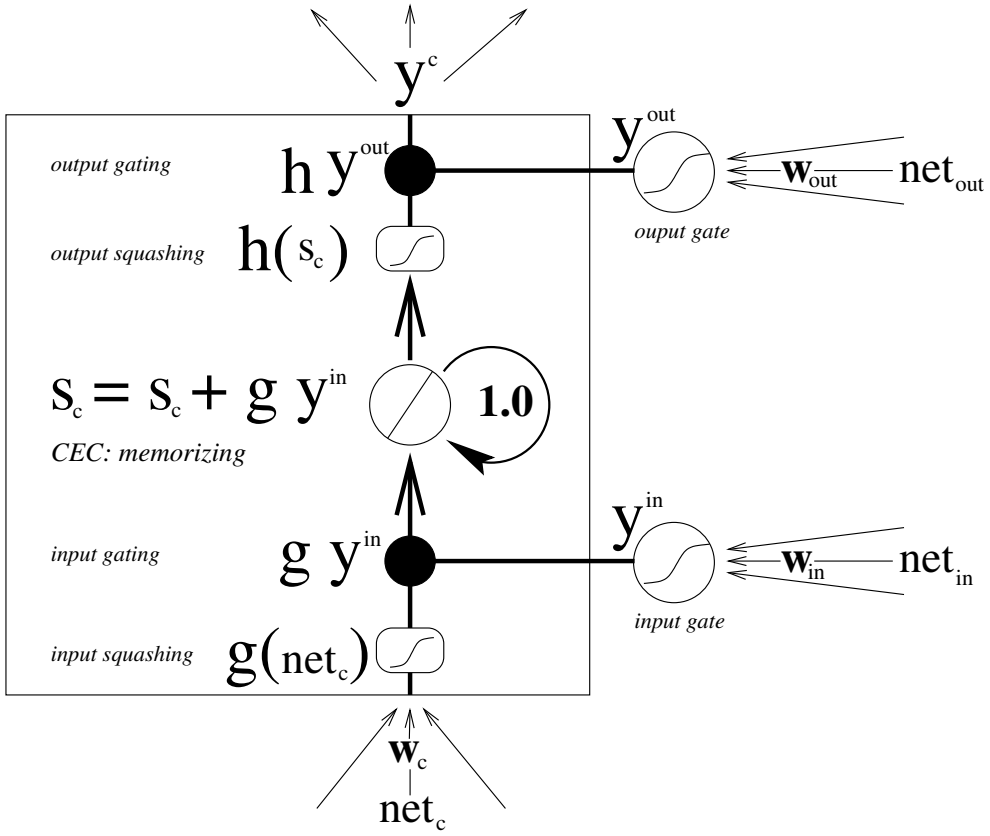

Figure 2.2: The traditional LSTM cell has a linear unit with a recurrent self-connection with weight 1.0 (CEC). Input and output gate regulate read and write access to the cell whose state is denoted s_c. The function g squashes the cell's input; h squashes the cell's output (see text for details).

error signals to the cell, the CEC's local error back flow remains constant, neither growing nor decaying. 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. Figure 2.2 shows a memory block with a single cell.

2.1 Forward Pass

The cell state, s_c, is updated based on its current state and three sources of input: net_c is input to the cell itself while 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:

$$net_{out_j}(t) = \sum_m w_{out_j m}\ y^m(t-1)\ ;\quad y^{out_j}(t) = f_{out_j}(net_{out_j}(t))\ , \tag{2.1}$$

$$net_{in_j}(t) = \sum_m w_{in_j m}\ y^m(t-1)\ ;\quad y^{in_j}(t) = f_{in_j}(net_{in_j}(t))\ . \tag{2.2}$$

Throughout this thesis j indexes memory blocks; v indexes memory cells in block j (with S_j cells), such that c_j^v denotes the v-th cell of the j-th memory block; 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 (if a source unit activation $y^m(t-1)$ refers to an input unit, current external input $y^m(t)$ is used instead). For the gates, f is a logistic sigmoid (with range $[0,1]$):

$$f(x) = \frac{1}{1+\mathrm{e}^{-x}} \quad . \tag{2.3}$$

The input to the cell itself is

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m}\ y^m(t-1) \quad , \tag{2.4}$$

which is is squashed by g, a centered logistic sigmoid function with range $[-2,2]$ (if not specified differently):

$$g(x) = \frac{4}{1+\mathrm{e}^{-x}} - 2 \quad . \tag{2.5}$$

The internal state of memory cell $s_c(t)$ is calculated by adding the squashed, gated input to the state at the last time step $s_c(t-1)$:

$$s_{c_j^v}(0) = 0 \ ; \quad s_{c_j^v}(t) = s_{c_j^v}(t-1) + y^{in_j}(t)\ g(net_{c_j^v}(t)) \quad \text{for } t > 0 \ . \tag{2.6}$$

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}:

$$y^{c_j^v}(t) = y^{out_j}(t)\ h(s_{c_j^v}(t)) \quad . \tag{2.7}$$

h is a centered sigmoid with range $[-1,1]$:

$$h(x) = \frac{2}{1+\mathrm{e}^{-x}} - 1 \quad . \tag{2.8}$$

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:

$$net_k(t) = \sum_m w_{km}\ y^m(t-1) \ , \quad y^k(t) = f_k(net_k(t)) \quad , \tag{2.9}$$

where m ranges over all units feeding the output units (typically all cells in the hidden layer, the input units, but not the memory block gates). As squashing function f_k we again use the logistic sigmoid (2.3). This concludes traditional LSTM's forward pass.

2.2 Learning

See Hochreiter & Schmidhuber (1997) for details of traditional LSTM's backward pass. It will be re-derived and discussed in detail in Section 3.2.2 after the introduction of forget gates.

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. In essence, once an error signal arrives at a memory cell output, it gets scaled by the output gate and the output nonlinearity h; then it enters the memory cell's linear CEC, where it can flow back indefinitely without ever being changed (this is why LSTM can bridge arbitrary time lags between input events and target signals). Only when the error escapes from the memory cell through an opening 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 consequence of this truncation is that each LSTM block relies on errors from the output for its adaptation. Since blocks do not exchange error signals, it is hard for LSTM to learn tasks where one block exclusively serves other blocks (e.g., as a pointer into a FIFO queue) without directly reducing the output error.

2.3 Tasks Solved with Traditional LSTM

Hochreiter and Schmidhuber (1997) already solved a wide range of tasks with traditional LSTM: (1) The embedded Reber grammar (a popular regular grammar benchmark); (2) Noise free and noisy sequences with time lags of up to 1000 steps (e.g.; the "2-sequence problem" proposed by Bengio et al., 1994); (3) Continuous-valued tasks the require the storage of values for long time periods and their summation and multiplication (up to a certain precision); (4) Temporal order problems with wildly separated inputs.

In the following chapters, however, we will present tasks (partly derived form the tasks listed above) on which traditional LSTM fails and point out its problems.

Chapter 3

Learning to Forget: Continual Prediction with LSTM

3.1 Introduction

Hochreiter and Schmidhuber (1997) demonstrated that LSTM can solve numerous tasks not solvable by previous learning algorithms for RNNs. In this chapter, however, we will show that even LSTM fails to learn to correctly process certain very long or continual time series that are not *a priori* segmented into appropriate training subsequences with clearly defined beginnings and ends at which the network's internal state could be reset. The problem is that a continual input stream eventually may cause the internal values of the cells to grow without bound, even if the repetitive nature of the problem suggests they should be reset occasionally. In this chapter will present a remedy.

While we present a specific solution to the problem of forgetting in LSTM networks, we recognize that *any* training procedure for RNNs which is powerful enough to span long time lags must also address the issue of forgetting in short term memory (unit activations). We know of no other current training method for RNNs which is sufficiently powerful to have encountered this problem.

Outline. Section 3.1.1 explains LSTM's weakness in processing continual input streams. Section 3.2 introduces a remedy called "forget gates." Forget gates learn to reset memory cell contents once they are not needed any more. Forgetting may occur rhythmically or in an input-dependent fashion. In the same section we derive a gradient-based learning algorithm for the LSTM extension with forget gates. Section 3.3 describes experiments: we transform well-known benchmark problems into more complex, continual tasks, report the performance of various RNN algorithms, and analyze and compare the networks found by traditional LSTM and extended LSTM.

3.1.1 Limits of traditional LSTM

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). If we present a continuous input stream, the cell states may grow in unbounded fashion, causing saturation of the output squashing function, h. This happens even if the nature of the problem suggests that the cell states should be reset occasionally, e.g., at the beginnings of new input sequences (whose starts, however, are not explicitly indicated by a teacher). 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, so that the cell will cease functioning as a memory. The problem did not arise in the experiments reported by Hochreiter & Schmidhuber (1997) because cell states were explicitly reset to zero before the start of each new sequence.

How can we solve this problem without losing LSTM's advantages over time delay neural networks (TDNN) (Waibel, 1989) or NARX networks (Lin et al., 1996), which depend on *a priori* knowledge of typical time lag sizes?

The standard technique of weight decay, which helps to contain the level of overall activity within the network, was found to generate solutions which were particularly prone to unbounded state growth.

Variants of focused back-propagation (Mozer, 1989) also do not work well. These let the internal state decay via a self-connection whose weight is smaller than 1. But there is no principled way of designing appropriate decay constants: A potential gain for some tasks is paid for by a loss of ability to deal with arbitrary, unknown causal delays between inputs and targets. In fact, state decay does not significantly improve experimental performance (see "State Decay" in Table 3.2).

Of course we might try to "teacher force" (Jordan, 1986; Doya & Yoshizawa, 1989) the internal states s_c by resetting them once a new training sequence starts. But this requires an external teacher who knows how to segment the input stream into training subsequences. We are precisely interested, however, in those situations where there is no *a priori* knowledge of this kind.

3.2 Solution: Forget Gates

Our solution to the problem above is to use adaptive "forget gates" which learn to reset memory blocks once their contents are out of date and hence useless. By resets we do not only mean immediate resets to zero but also gradual resets corresponding to slowly fading cell states.

More specifically, we replace traditional LSTM's constant CEC weight 1.0 by the multiplicative forget gate activation y^{φ}. See Figure 3.1.

3.2.1 Forward Pass of Extended LSTM with Forget Gates

All equations of traditional LSTM's forward pass except for equation (2.6) will remain valid also for extended LSTM with forget gates.

The forget gate activation y^{φ} is calculated like the activations of the other gates—see equations (2.1) and (2.2):

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m}\, y^m(t-1)\ ; \quad y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t))\ . \tag{3.1}$$

Here net_{φ_j} is the input from the network to the forget gate. We use the logistic sigmoid with range $[0, 1]$ as squashing function f_{φ_j}. Its output becomes the weight of the self recurrent

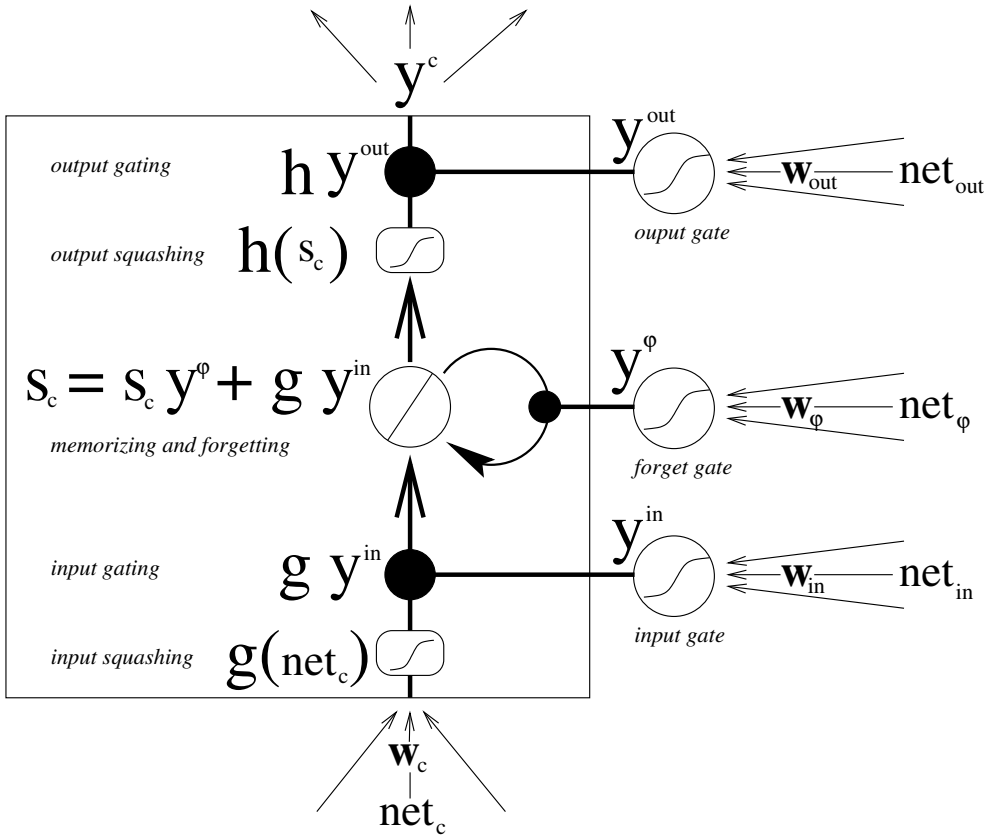

Figure 3.1: Memory block with only one cell for the extended LSTM. A multiplicative forget gate can reset the cell's inner state s_c.

connection of the internal state s_c in equation (2.6). The revised update equation for s_c in the extended LSTM algorithm is (for $t > 0$):

$$s_{c_j^v}(t) \quad = \quad y^{\varphi_j}(t)\ s_{c_j^v}(t-1) + y^{in_j}(t)\ \ g(net_{c_j^v}(t)) \quad , \tag{3.2}$$

with $s_{c_j^v}(0) = 0$. Extended LSTM's full forward pass is obtained by adding equations (3.1) to those in Chapter 2 and replacing equation (2.6) by (3.2).

Bias weights for LSTM gates are initialized with negative values for input and output gates (see Section 3.3.2), positive values for forget gates. This implies—compare equations (3.1) and (3.2)—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 traditional LSTM cell. It will not explicitly forget anything until it has learned to forget.

3.2.2 Backward Pass of Extended LSTM with Forget Gates

LSTM's backward pass is an efficient fusion of slightly modified, truncated back propagation through time (BPTT) (e.g Williams & Peng 1990) and a customized version of real time recurrent learning (RTRL) (e.g. Robinson & Fallside 1987). Output units use BP; output gates use slightly modified, truncated BPTT. Weights to cells, input gates and the novel forget gates, however, use a truncated version of RTRL. Truncation means that all errors are cut off once they leak out of a memory cell or gate, although they do serve to change the incoming

weights. The effect is that the CECs are the only part of the system through which errors can flow back forever. This makes LSTM's updates efficient without significantly affecting learning power: error flow outside of cells tends to decay exponentially anyway (Hochreiter, 1991). In the equations below, $\stackrel{tr}{=}$ will indicate where we use error truncation and, for simplicity, unless otherwise indicated, we assume only a single cell per block.
We start with the usual squared error objective function based on targets t^k:

$$E(t) = \frac{1}{2} \sum_k e_k(t)^2 \;\; ; \;\; e_k(t) := t^k(t) - y^k(t) \;\; , \tag{3.3}$$

where e_k denotes the externally injected error. We minimize E via gradient descent by adding weight changes Δw_{lm} to the weights w_{lm} (from unit m to unit l) using learning rate α (δ_{ij} is the Kronecker delta):

$$\begin{aligned}
\Delta w_{lm}(t) &= -\alpha \, \frac{\partial E(t)}{\partial w_{lm}} = -\alpha \sum_k \frac{\partial E(t)}{\partial y^k(t)} \, \frac{\partial y^k(t)}{\partial w_{lm}} = \alpha \sum_k e_k(t) \, \frac{\partial y^k(t)}{\partial w_{lm}} \\
&= \alpha \sum_k \sum_{l'} e_k(t) \, \frac{\partial y^k(t)}{\partial y^{l'}(t)} \, \frac{\partial y^{l'}(t)}{\partial net_{l'}(t)} \, \frac{\partial net_{l'}(t)}{\partial w_{lm}} \\
&= \alpha \sum_k \sum_{l'} e_k(t) \, \frac{\partial y^k(t)}{\partial y^{l'}(t)} \, \frac{\partial y^{l'}(t)}{\partial net_{l'}(t)} \left(\delta_{l'l} \, y^m(t-1) + \frac{\partial net_{l'}(t)}{\partial y^m(t-1)} \right)
\end{aligned}$$

Errors are truncated when they leave a memory block by setting the following derivatives in the above equation to zero: $\frac{\partial net_{l'}(t)}{\partial y_m(t-1)} \stackrel{tr}{=} 0$ for $l' \in \{\varphi, in, c_j^v\}$.

$$\begin{aligned}
\Delta w_{lm}(t) &\stackrel{tr}{=} \alpha \sum_k e_k(t) \, \frac{\partial y^k(t)}{\partial y^l(t)} \, \frac{\partial y^l(t)}{\partial net_l(t)} \, y^m(t-1) \\
&= \alpha \underbrace{\frac{\partial y^l(t)}{\partial net_l(t)} \left(\sum_k \frac{\partial y^k(t)}{\partial y^l(t)} e_k(t) \right)}_{=: \, \delta_l(t)} y^m(t-1) \;\; .
\end{aligned} \tag{3.4}$$

For an arbitrary output unit ($l = k'$) the sum in (3.4) reduces to e_k (with $k = k'$). By differentiating equation (2.9) we obtain the usual back-propagation weight changes for the output units:

$$\frac{\partial y^k(t)}{\partial net_k(t)} = f_k'(net_k(t)) \implies \delta_k(t) = f_k'(net_k(t)) \; e_k(t) \;\; . \tag{3.5}$$

To compute the weight changes for the output gates $\Delta w_{out_j m}$ we set ($l = out$) in (3.4). The resulting terms can be determined by differentiating equations (2.1), (2.7) and (2.9):

$$\frac{\partial y^{out_j}(t)}{\partial net_{out_j}(t)} = f_{out_j}'(net_{out_j}(t)) \;\; , \;\; \frac{\partial y^k(t)}{\partial y^{out_j}(t)} \, e_k(t) = h(s_{c_j^v}(t)) \; w_{kc_j^v} \; \delta_k(t) \;\; .$$

Inserting both terms in equation (3.4) gives $\delta_{out_j}^v$, the contribution of the block's v-th cell to δ_{out_j}. As every cell in a memory block contributes to the weight change of the output gate, we

have to sum over all cells v in block j to obtain the total δ_{out_j} of the j-th memory block (with S_j cells):

$$\delta_{out_j}(t) = f'_{out_j}(net_{out_j}(t)) \left(\sum_{v=1}^{S_j} h(s_{c_j^v}(t)) \sum_k w_{kc_j^v} \ \delta_k(t) \right) \quad . \tag{3.6}$$

Equations (3.4), (3.5)and (3.6) define the weight changes for output units and output gates of memory blocks. Their derivation was almost standard BPTT, with error signals truncated once they leave memory blocks (including its gates). This truncation does not affect LSTM's long time lag capabilities but is crucial for all equations of the backward pass and should be kept in mind.

For weights to cell, input gate and forget gate we adopt an RTRL-oriented perspective, by first stating the influence of a cell's internal state $s_{c_j^v}$ on the error and then analyzing how each weight to the cell or the block's gates contributes to $s_{c_j^v}$. So we split the gradient in a way different from the one used in equation (3.4), neglecting, however, the same derivatives:

$$\Delta w_{lm}(t) = -\alpha \ \frac{\partial E(t)}{\partial w_{lm}} \stackrel{tr}{=} -\alpha \ \underbrace{\frac{\partial E(t)}{\partial s_{c_j^v}(t)}}_{=: \ -e_{s_{c_j^v}}(t)} \ \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} = \alpha \ e_{s_{c_j^v}}(t) \ \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \ . \tag{3.7}$$

These terms are the internal state error $e_{s_{c_j^v}}$ and a 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 ($l = \varphi$), as all these weights contribute to the calculation of $s_{c_j^v}(t)$. We treat the partial for the internal states error $e_{s_{c_j^v}}$ analogously to (3.4) and obtain:

$$e_{s_{c_j^v}}(t) := -\frac{\partial E(t)}{\partial s_{c_j^v}(t)} \stackrel{tr}{=} -\frac{\partial E(t)}{\partial y^k(t)} \ \frac{\partial y^k(t)}{\partial y^{c_j^v}(t)} \ \frac{\partial y^{c_j^v}(t)}{\partial s_{c_j^v}(t)} = \frac{\partial y^{c_j^v}}{\partial s_{c_j^v}(t)} \ \sum_k \underbrace{\frac{\partial y^k(t)}{\partial y^{c_j^v}(t)} \ e_k(t)}_{= w_{c_j^v l} \ \delta_k(t)}$$

Differentiating the forward pass equation (2.7), we obtain:

$$\frac{\partial y^{c_j^v}}{\partial s_{c_j^v}(t)} = y^{out_j}(t) \ \ h'(s_{c_j^v}(t)) \quad .$$

Substituting this term in the equation for $e_{s_{c_j^v}}$:

$$e_{s_{c_j^v}}(t) \ = \ y^{out_j}(t) \ \ h'(s_{c_j^v}(t)) \left(\sum_k w_{kc_j^v} \ \delta_k(t) \right) \quad . \tag{3.8}$$

This internal state error needs to be calculated for each memory cell. To calculate the partial $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ in equation (3.7) we differentiate equation (3.2) and obtain a sum of four terms.

$$\begin{aligned} \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \ = \ & \underbrace{\frac{\partial s_{c_j^v}(t-1)}{\partial w_{lm}} \ y^{\varphi_j}(t)}_{\neq 0 \text{ for all } l \in \{\varphi, in, c_j^v\}} \ + \ \underbrace{y^{in_j}(t) \ \frac{\partial g(net_{c_j^v}(t))}{\partial w_{lm}}}_{\neq 0 \text{ for } l = c_j^v \text{ (cell)}} \\ & + \ \underbrace{g(net_{c_j^v}(t)) \ \frac{\partial y^{in_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = in \text{ (input gate)}} \ + \ \underbrace{s_{c_j^v}(t-1) \ \frac{\partial y^{\varphi_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = \varphi \text{ (forget gate)}} \quad . \end{aligned} \tag{3.9}$$

Differentiating the forward pass equations (2.6), (2.2) and (3.1) for g, y^{in}, and y^{φ} we can substitute the unresolved partials and split the expression on the right hand side of (3.9) into three separate equations for the cell ($l = c_j^v$), the input gate ($l = in$) and the forget gate ($l = \varphi$):

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} \, y^{\varphi_j}(t) + g'(net_{c_j^v}(t)) \; y^{in_j}(t) \; y^m(t-1) \; , \tag{3.10}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} \, y^{\varphi_j}(t) + g(net_{c_j^v}(t)) \; f'_{in_j}(net_{in_j}(t)) \; y^m(t-1) \; , \tag{3.11}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} \, y^{\varphi_j}(t) + s_{c_j^v}(t-1) \; f'_{\varphi_j}(net_{\varphi_j}(t)) \; y^m(t-1) \; . \tag{3.12}$$

Furthermore the initial state of network does not depend on the weights, so we have

$$\frac{\partial s_{c_j^v}(t=0)}{\partial w_{lm}} = 0 \quad \text{for} \;\; l \in \{\varphi, in, c_j^v\} \;\; . \tag{3.13}$$

Note that the recursions in equations (3.10)-(3.12) depend on the actual activation of the block's forget gate. When the activation goes to zero not only the cell's state, but also the partials are reset (forgetting includes forgiving previous mistakes). Every cell needs to keep a copy of each of these three partials and update them at every time step.

We can insert the partials in equation (3.7) and calculate the corresponding weight updates, with the internal state error $e_{s_{c_j^v}}(t)$ given by equation (3.8). The difference between updates of weights to a cell itself ($l = c_j^v$) and updates of weights to the gates is that changes to weights to the cell $\Delta w_{c_j^v m}$ only depend on the partials of this cell's own state:

$$\Delta w_{c_j^v m}(t) = \alpha \; e_{s_{c_j^v}}(t) \; \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \;\; . \tag{3.14}$$

To update the weights of the input gate and of the forget gate, however, we have to sum over the contributions of all cells in the block:

$$\Delta w_{lm}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \;\; \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \quad \text{for} \;\; l \in \{\varphi, in\} \;\; . \tag{3.15}$$

The equations necessary to implement the backward pass are (3.4), (3.5), (3.6), (3.8), (3.10), (3.11), (3.12), (3.13), (3.14) and (3.15).

3.2.3 Complexity

To calculate the computational complexity of extended LSTM we take into account that weights to input gates and forget gates cause more expensive updates than others, because each such weight directly affects all the cells in its memory block. We evaluate a rather typical topology used in the experiments (see Figure 3.3). All memory blocks have the same size; gates have no outgoing connections; output units and gates have a bias connection (from a unit whose activation is always 1.0); other connections to output units stem from memory blocks only; the hidden layer is fully connected. Let B, S, I, K denote the numbers of memory blocks, memory

cells in each block, input units, and output units, respectively. We find the update complexity per time step to be:

$$\begin{aligned} W_c &= \underbrace{B \cdot [S \cdot (\overbrace{B \cdot S + 1}^{\text{to cells}} + \overbrace{2 \cdot (B \cdot S + 1)}^{\text{to input and forget gates}}) + \overbrace{B \cdot S + 1}^{\text{to output gate}}]}_{\text{recurrent connections and bias}} \\ &+ \underbrace{K \cdot (B \cdot S + 1)}_{\text{to output}} + \underbrace{I \cdot (B \cdot (S + 2 \cdot S + 1))}_{\text{from input}} \\ &= O(B^2 \cdot S^2) + O(K \cdot B \cdot S) + O(I \cdot B \cdot S) \quad , \end{aligned} \tag{3.16}$$

Keeping K and I fixed we obtain a total computational complexity of $O(B^2 \cdot S^2)$. The number of weights is:

$$N_w = \underbrace{B \cdot [\overbrace{S \cdot (B \cdot S + 1)}^{\text{to cells}} + \overbrace{3 \cdot (B \cdot S + 1)}^{\text{to gates}})]}_{\text{recurrent connections and bias}} + \underbrace{K \cdot (B \cdot S + 1)}_{\text{to output}} + \underbrace{I \cdot (B \cdot S + 3 \cdot B))}_{\text{from input}} ;$$

with K and I fixed:

$$N_w = O(B^2 \cdot S^2) \quad .$$

Hence LSTM's computational complexity per time step and weight is $O(1)$. Considering connections to gates separately we find that their computational complexity per time step and weight is $O(S)$. But this is compensated by the "less complex" connections to the cells of $O(1)$. It is essentially the same as for a fully connected BPTT recurrent network. Storage complexity per weight is also $O(1)$, as the last time step's partials from equations (3.10), (3.11) and (3.12) are all that need to be stored for the backward pass. So the storage complexity does not depend on the length of the input sequence. Hence extended LSTM is local in space and time, according to Schmidhuber's definition (1989), just like traditional LSTM.

3.3 Experiments

3.3.1 Continual Embedded Reber Grammar Problem

To generate an infinite input stream we extend the well-known "embedded Reber grammar" (ERG) benchmark problem, e.g., Smith and Zipser (1989), Cleeremans et al. (1989), Fahlman (1991), Hochreiter & Schmidhuber (1997). Consider Figure 3.2.

ERG. The traditional method starts at the leftmost node of the ERG graph, and sequentially generates finite symbol strings (beginning with the empty string) by steping from node to node following the edges of the graph, and appending the symbols associated with the edges to the current string until the rightmost node is reached. Edges are chosen randomly if there is a choice (probability = 0.5).

Input and target symbols are represented by 7 dimensional binary vectors, each component standing for one of the 7 possible symbols. Hence the network has 7 input units and 7 output units. The task is to read strings, one symbol at a time, and to continually predict the next possible symbol(s). Input vectors have exactly one nonzero component. Target vectors may have two, because sometimes there is a choice of two possible symbols at the next step. A prediction is considered correct if the error at each of the 7 output units is below 0.49 (error signals occur at every time step).

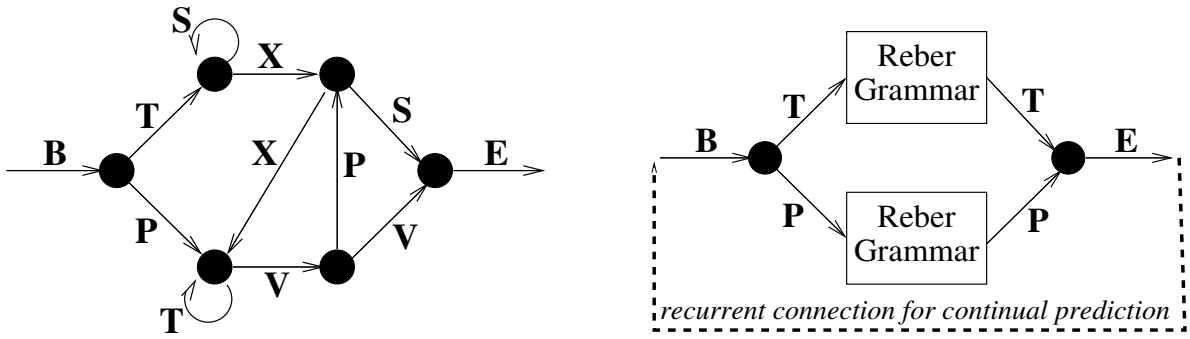

Figure 3.2: Transition diagrams for standard (left) and embedded (right) Reber grammars. The dashed line indicates the continual variant.

Algo-rithm	# hidden units	#weights	learning rate	% of success	success after
RTRL	3	$\approx$ 170	0.05	"some fraction"	173,000
RTRL	12	$\approx$ 494	0.1	"some fraction"	25,000
ELM	15	$\approx$ 435		0	>200,000
RCC	7-9	$\approx$ 119-198		50	182,000
Tra. LSTM	3bl.,size 2	276	0.5	100	8,440

Table 3.1: Standard embedded Reber grammar (ERG): percentage of successful trials and number of sequence presentations until success for RTRL (results taken from Smith and Zipser 1989), "Elman net trained by Elman's procedure" (results taken from Cleeremans et al. 1989), "Recurrent Cascade-Correlation" (results taken from Fahlman 1991) and traditional LSTM (results taken from Hochreiter and Schmidhuber 1997). Weight numbers in the first 4 rows are estimates.

To correctly predict the symbol before the last (**T** or **P**) in an ERG string, the network has to remember the second symbol (also **T** or **P**) without confusing it with identical symbols encountered later. The minimal time lag is 7 (at the limit of what standard recurrent networks can manage); time lags have no upper bound though. The expected length of a string generated by an ERG is 11.5 symbols. The length of the longest string in a set of N non-identical strings is proportional to $\log N$ (statistics of the embedded Reber Grammar are discussed in Appendix A). For the training and test sets used in our experiments, the expected value of the longest string is greater than 50.

Table 3.1 summarizes performance of previous RNNs on the standard ERG problem (testing involved a test set of 256 ERG test strings). Only traditional LSTM always learns to solve the task. Even when we ignore the unsuccessful trials of the other approaches, LSTM learns much faster.

CERG. Our more difficult continual variant of the ERG problem (CERG) does not provide information about the beginnings and ends of symbol strings. Without intermediate resets, the network is required to learn, in an on-line fashion, from input streams consisting of concatenated ERG strings. Input streams are stopped as soon as the network makes an incorrect prediction or the 10^5-th successive symbol has occurred. Learning and testing alternate: after each training

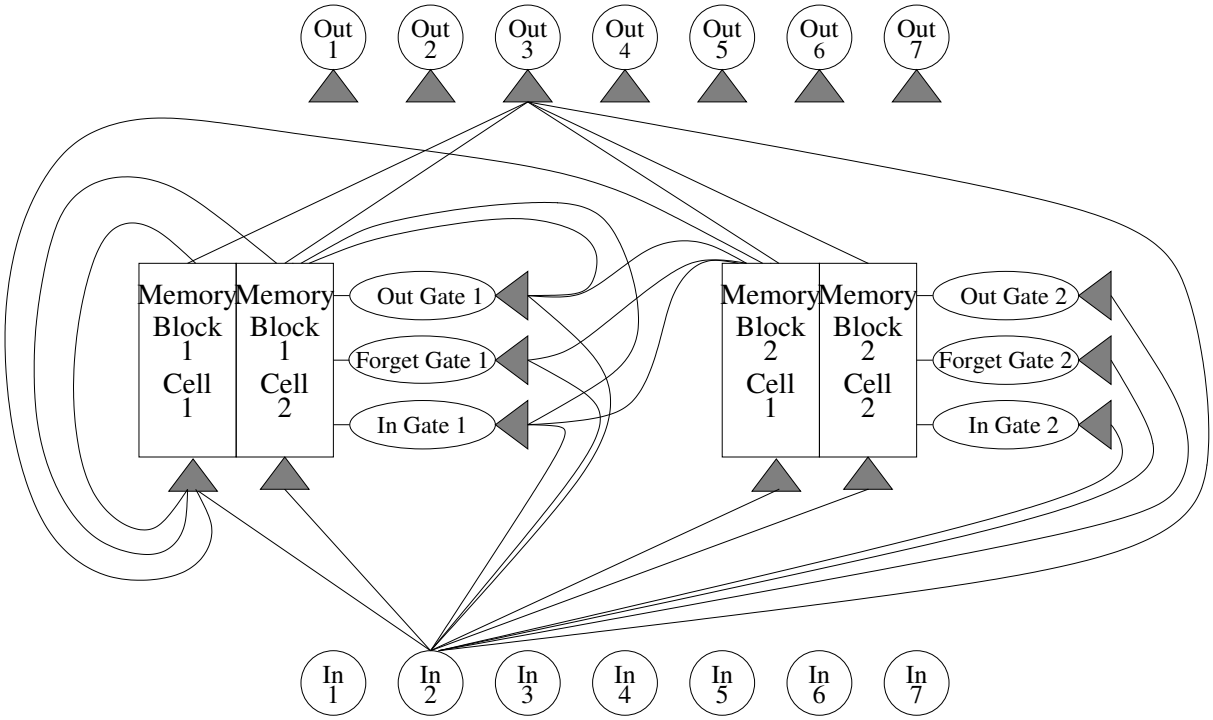

Figure 3.3: Three layer LSTM topology with recurrence limited to the hidden layer consisting of four extended LSTM memory blocks (only two shown) with two cells each. Only a limited subset of connections are shown.

stream we freeze the weights and feed 10 test streams. Our performance measure is the average test stream size; 100,000 corresponds to a so-called "perfect" solution (10^6 successive correct predictions).

3.3.2 Network Topology and Parameters

The 7 input units are fully connected to a hidden layer consisting of 4 memory blocks with 2 cells each (8 cells and 12 gates in total). The cell outputs are fully connected to the cell inputs, to all gates, and to the 7 output units. The output units have additional "shortcut" connections from the input units (see Figure 3.3). All gates and output units are biased. Bias weights to in- and output gates are initialized blockwise: -0.5 for the first block, -1.0 for the second, -1.5 for the third, and so forth. In this manner, cell states are initially close to zero, and, as training progresses, the biases become progressively less negative, allowing the serial activation of cells as active participants in the network computation. Forget gates are initialized with symmetric positive values: $+0.5$ for the first block, $+1$ for the second block, etc. Precise bias initialization is not critical though—other values work just as well. All other weights including the output bias are initialized randomly in the range $[-0.2, 0.2]$. There are 424 adjustable weights, which is comparable to the number used by LSTM in solving the ERG (see Table 3.1).

Weight changes are made after each input symbol presentation. At the beginning of each training stream, the learning rate α is initialized with 0.5. It either remains fixed or decays by a factor of 0.99 per time step (LSTM with α-decay). Learning rate decay is well studied in statistical approximation theory and is also common in neural networks, e.g. (Darken, 1995).

Algorithm	%Solutions	%Good Sol.	%Rest
Tra. LSTM with external reset	74 (7441)	0 ⟨−⟩	26 ⟨31⟩
Traditional LSTM	0 (-)	1 ⟨1166⟩	99 ⟨37⟩
LSTM with State Decay (0.9)	0 (-)	0 ⟨−⟩	100 ⟨56⟩
LSTM with Forget Gates	18 (18889)	29 ⟨39171⟩	53 ⟨145⟩
LSTM with Forget Gates and sequential α decay	62 (14087)	6 ⟨68464⟩	32 ⟨30⟩

Table 3.2: Continuous Embedded Reber Grammar (CERG): Column "%Solutions": Percentage of "perfect" solutions (correct prediction of 10 streams of 100,000 symbols each), in parenthesis the number of training streams presented until solution was reached. Column "Good Sol.": Percentage of solutions with an average stream length > 1000 (mean length of error free prediction is given in angle brackets). Column "Rest": percentage of "bad" solutions with average stream length ≤ 1000 (mean length of error free prediction is given in angle brackets). The results are averages over 100 independently trained networks. Other algorithms like BPTT are not included in the comparison, because they tend to fail even on the easier, non-continual ERG.

We report results of exponential α-decay (as specified above), but also tested several other variants (linear, $1/T$, $1/\sqrt{T}$), and found them all to work as well without extensive optimization of parameters.

3.3.3 CERG Results

Training was stopped after at most 30000 training streams, each of which was ended when the first prediction error or the 100000th successive input symbol occurred. Table 3.2 compares extended LSTM (with and without learning rate decay) to traditional LSTM and an LSTM variant with decay of the internal cell state s_c (with a self recurrent weight < 1). Our results for traditional LSTM with network activation resets (by an external teacher) at sequence ends are slightly better than those based on a different topology (Hochreiter & Schmidhuber, 1997). External resets (non-continual case) allow LSTM to find excellent solutions in 74% of the trials, according to our stringent testing criterion. Traditional LSTM fails, however, in the continual case. Internal state decay does not help much either (we tried various self-recurrent weight values and report only the best result). Extended LSTM with forget gates, however, can solve the continual problem.

A continually decreasing learning rate led to even better results but had no effect on the other algorithms. Different topologies may provide better results, too—we did not attempt to optimize topology.

Can the network learn to recognize appropriate times for opening/closing its gates without using the information conveyed by the marker symbols **B** and **E**? To test this we replaced all CERG subnets of the type $\overset{\mathbf{T\backslash P}}{\longrightarrow} \bullet \overset{\mathbf{E}}{\longrightarrow} \bullet \overset{\mathbf{B}}{\longrightarrow} \bullet \overset{\mathbf{T\backslash P}}{\longrightarrow}$ by $\overset{\mathbf{T\backslash P}}{\longrightarrow} \bullet \overset{\mathbf{T\backslash P}}{\longrightarrow}$.

This makes the task more difficult as the net now needs to keep track of sequences of numerous potentially confusing **T** and **P** symbols. But LSTM with forget gates (same topology) was still able to find perfect solutions, although less frequently (sequential α decay was not applied).

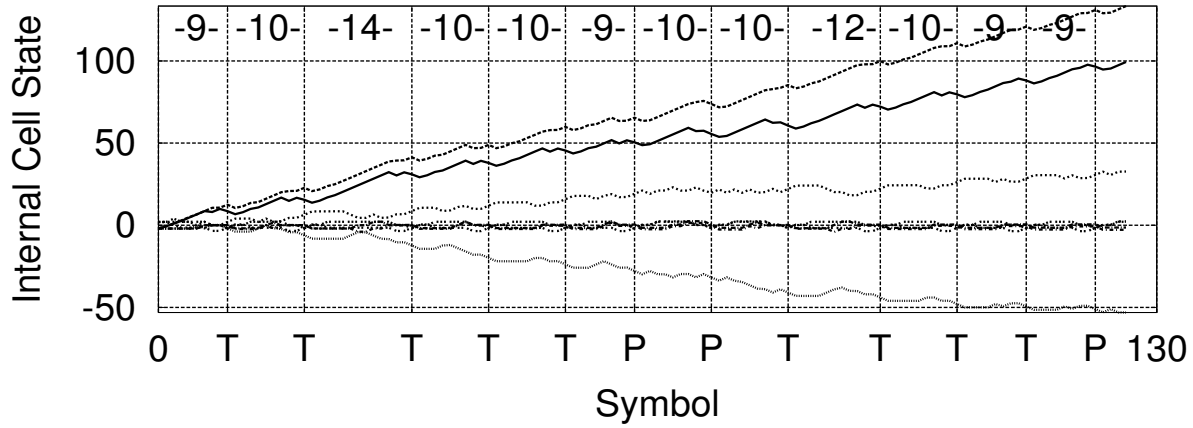

Figure 3.4: Evolution of traditional LSTM's internal states s_c during presentation of a test stream stopped at first prediction failure. Starts of new ERG strings are indicated by vertical lines labeled by the symbols (**P** or **T**) to be stored until the next string start.

3.3.4 Analysis of the CERG Results

How does extended LSTM solve the task on which traditional LSTM fails? Section 3.1.1 already mentioned LSTM's problem of uncontrolled growth of the internal states. Figure 3.4 shows the evolution of the internal states s_c during the presentation of a test stream. The internal states tend to grow linearly. At the starts of successive ERG strings, the network is in an increasingly active state. At some point (here after 13 successive strings), the high level of state activation leads to saturation of the cell outputs, and performance breaks down. Extended LSTM, however, learns to use the forget gates for resetting its state when necessary. Figure 3.5 (top half) shows a typical internal state evolution after learning. We see that the third memory block resets its cells in synchrony with the starts of ERG strings (the vertical lines in Figure 3.5 indicate the third symbol of a string). The internal states oscillate around zero; they never drift out of bounds as with traditional LSTM (Figure 3.4). It also becomes clear how the relevant information gets stored: the second cell of the third block stays negative while the symbol **P** has to be stored, whereas a **T** is represented by a positive value. The third block's forget gate activations are plotted in Figure 3.5 (bottom). Most of the time they are equal to 1.0, thus letting the memory cells retain their internal values. At the end of an ERG string the forget gate's activation goes to zero, thus resetting cell states to zero.

Analyzing the behavior of the other memory blocks, we find that only the third is directly responsible for bridging ERG's longest time lag (which is sufficient as one just bit has to be stored). Figure 3.6 plots values analogous to those in Figure 3.5 for the first memory block and its first cell. The first block's cell and forget gate show short-term behavior only (necessary for predicting the numerous short time lag events of the Reber grammar). The same is true for all other blocks except the third. Common to all memory blocks is that they learned to reset themselves in an appropriate fashion.

3.3.5 Continual Noisy Temporal Order Problem

Extended LSTM solves the CERG problem while traditional LSTM does not. But can traditional LSTM solve problems which extended LSTM cannot? We tested extended LSTM on one of the

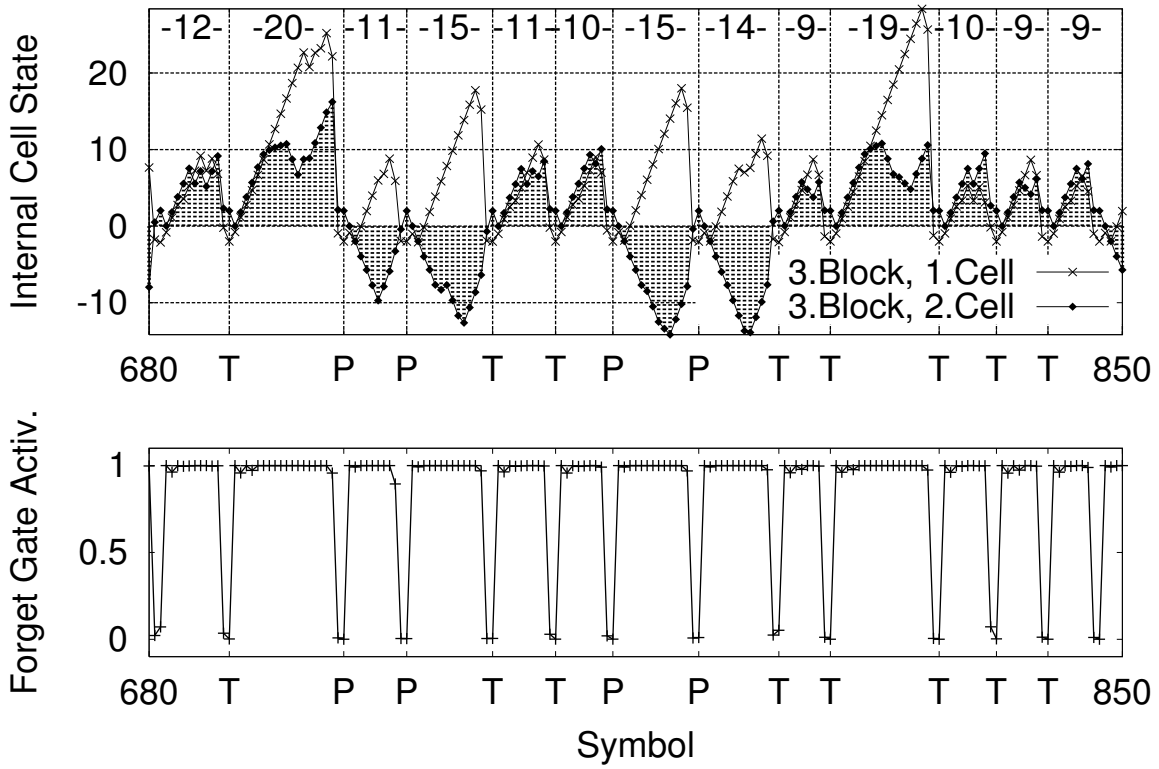

Figure 3.5: Top: Internal states s_c of the two cells of the self-resetting third memory block in an extended LSTM network during a test stream presentation. The figure shows 170 successive symbols taken from the longer sequence presented to a network that learned the CERG. Starts of new ERG strings are indicated by vertical lines labeled by the symbols (**P** or **T**) to be stored until the next string start. Bottom: simultaneous forget gate activations of the same memory block.

most difficult nonlinear long time lag tasks ever solved by an RNN: "Noisy Temporal Order" (NTO) (task 6b taken from Hochreiter & Schmidhuber 1997).

NTO. The goal is to classify sequences of locally represented symbols. Each sequence starts with an E, ends with a B (the "trigger symbol"), and otherwise consists of randomly chosen symbols from the set $\{a, b, c, d\}$ except for three elements at positions t_1, t_2 and t_3 that are either X or Y (Figure 3.7). The sequence length is randomly chosen between 100 and 110, t_1 is randomly chosen between 10 and 20, t_2 is randomly chosen between 33 and 43, and t_3 is randomly chosen between 66 and 76. There are 8 sequence classes Q, R, S, U, V, A, B, C which depend on the temporal order of the Xs and Ys. The rules are (temporal order $\rightarrow$ class): $X, X, X \rightarrow Q$; $X, X, Y \rightarrow R$; $X, Y, X \rightarrow S$; $X, Y, Y \rightarrow U$; $Y, X, X \rightarrow V$; $Y, X, Y \rightarrow A$; $Y, Y, X \rightarrow B$; $Y, Y, Y \rightarrow C$. Target signals occur only at the end of a sequence. The problem's minimal time lag size is 80 (!). Forgetting is only harmful as all relevant information has to be kept until the end of a sequence, after which the network is reset anyway.

We use the network topology described in section 3.3.2 with 8 input and 8 output units. Using a large bias (5.0) for the forget gates, extended LSTM solved the task as quickly as traditional LSTM (recall that a high forget gate bias makes extended LSTM degenerate into traditional LSTM). Using a moderate bias like the one used for CERG (1.0), extended LSTM

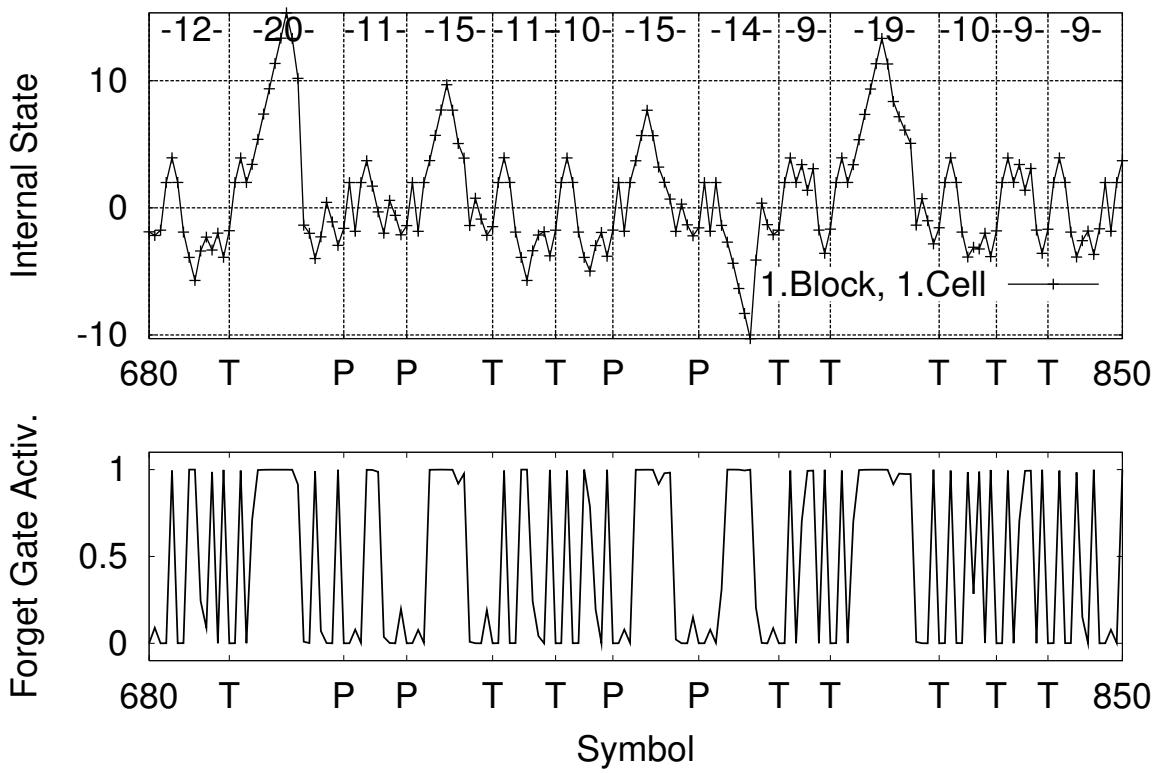

Figure 3.6: Top: Extended LSTM's self-resetting states for the first cell in the first block. Bottom: forget gate activations of the first memory block.

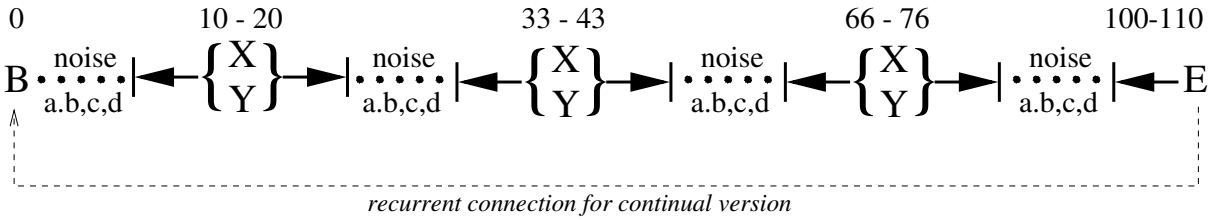

Figure 3.7: NTO and CNTO tasks. See text for details.

took about three times longer on average, but did solve the problem. The slower learning speed results from the net having to learn to remember everything and not to forget.
Generally speaking, we have not yet encountered a problem that LSTM solves while extended LSTM does not.

CNTO. Now we take the next obvious step and transform the NTO into a continual problem that does require forgetting, just as in section 3.3.1, by generating continual input streams consisting of concatenated NTO sequences (Figure 3.7). Processing such streams without intermediate resets, the network is required to learn to classify NTO sequences in an online fashion. Each input stream is stopped once the network makes an incorrect classification or 100 successive NTO sequences have been classified correctly. Learning and testing alternate; the performance measure is the average size of 10 test streams, measured by the number of their NTO sequences (each containing between 100 and 110 input symbols). Training is stopped after at most 10^5

Algorithm	%Perfect Sol.	%Partial Sol.
Traditional LSTM	0 (-)	100 ⟨4.6⟩
LSTM with Forget Gates	24 (18077)	76 ⟨12.2⟩
LSTM with Forget Gates and sequential α decay	37 (22654)	63 ⟨11.8⟩

Table 3.3: Continuous Noisy Temporal Order (CNTO): Column "%Perfect Sol.": Percentage of "perfect" solutions (correct classification of 1000 successive NTO sequences in 10 test streams); in parentheses: number of training streams presented. Column "%Partial Sol.": percentage of solutions and average stream size (value in angular brackets) ≤ 100. All results are averages over 100 independently trained networks. Other algorithms (BPTT, RTRL etc.) are not included in the comparison, because they fail even on the easier, non-continual NTO.

training streams.

Results. Table 3.3 summarizes the results. We observe that traditional LSTM again fails to solve the continual problem. Extended LSTM with forget gates, however, can solve it. A continually decreasing learning rate (α decaying by a fraction of 0.9 after each NTO sequence in a stream) leads to slightly better results but is not necessary.

3.4 Conclusion

Continual input streams generally require occasional resets of the stream-processing network. Partial resets are also desirable for tasks with hierarchical decomposition. For instance, reoccurring subtasks should be solved by the same network module, which should be reset once the subtask is solved. Since typical real-world input streams are not *a priori* decomposed into training subsequences, and since typical sequential tasks are not *a priori* decomposed into appropriate subproblems, RNNs should be able to *learn* to achieve appropriate decompositions. The novel forget gates naturally permit LSTM to learn local self-resets of memory contents that have become irrelevant.

LSTM extended with forget gates holds promise for any sequential processing task in which we suspect that a hierarchical decomposition may exist, but do not know in advance what this decomposition is. The model has been successfully applied to the task of discriminating languages from very limited prosodic information (Cummins, Gers, & Schmidhuber, 1999) where there is no clear linguistic theory of hierarchical structure.

Chapter 4

Arithmetic Operations on Continual Input Streams

4.1 Introduction

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.

Due to its architecture traditional LSTM is well suited for tasks involving addition, subtraction and integration (Hochreiter & 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.

4.2 Experiments

We focus on tasks involving arithmetic operations on input streams that so far have been addressed only in non-continual settings (Tsung & Cottrell, 1989; Hochreiter & Schmidhuber, 1997).

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_{2m-1} for $m = 1, 2, \ldots$, 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_{2m} 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_{2m-1} then the target at t_{2m} is the sum of the previous target at t_{2m-2} and the "marked" first input component at t_{2m-1}. Otherwise it is the product of these two values.

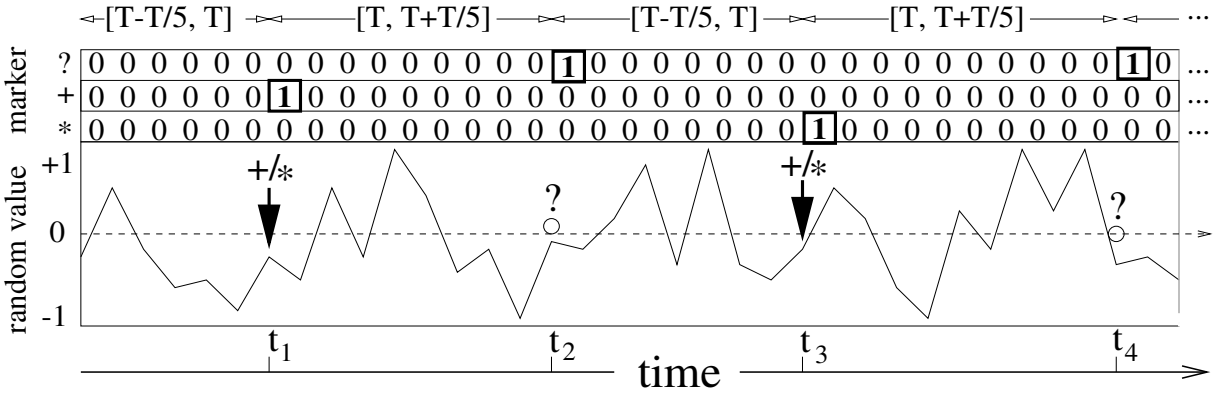

Figure 4.1: Illustration of the continual addition (and multiplication) tasks.

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 4.1 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_{2m-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_{2m+2} equal the sum of the two most recent marked values at times t_{2m+1} and t_{2m-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.

4.2.1 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 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.) 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.

4.2.2 Results

See Table 4.1. 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

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

Table 4.1: 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.

networks.

Task 1. Both traditional 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 traditional 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. Traditional 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 3.3.4), by resetting states once stored information becomes obsolete.

The results confirm that forget gates are mandatory for LSTM fed with continual input streams (Chapter 3)., where obsolete memories need to be discarded at some point (see "Task 3: Gliding addition"). Experiment 2 shows that forget gates also greatly facilitate operations involving multiplication.

4.3 Conclusion

In this chapter we demonstrated that forget gates do not only serve for the processing of continual input streams but also augment LSTM's arithmetic capabilities.

We presented tasks on continual input streams with a level of arithmetic complexity where traditional LSTM fails but LSTM with forget gates solves the tasks in an elegant way. On the other hand we have not found a task yet that traditional LSTM can solve but LSTM with forget gates cannot.

Chapter 5

Learning Precise Timing with Peephole LSTM

5.1 Introduction

Humans quickly learn to recognize rhythmic pattern sequences, whose defining aspects are the temporal intervals between sub-patterns. Conversely, drummers and others are also able to generate precisely timed rhythmic sequences of motor commands. This motivates the study of artificial systems that learn to separate or generate patterns that convey information through the length of intervals between events.

Widely used approaches to sequence processing, such as Hidden Markov Models (HMMs), typically discard such information. They are successful in speech recognition precisely because they do not care for the difference between slow and fast versions of a given spoken word. Other tasks such as rhythm detection, music processing, and the tasks in this chapter, however, do require exact time measurements. Although an HMM could deal with a finite set of intervals between given events by devoting a separate internal state for each interval, this would be cumbersome and inefficient, and would not use the very strength of HMMs to be invariant to non-linear temporal stretching.

RNNs hold more promise for recognizing patterns that are defined by temporal distance. In fact, while HMMs and traditional discrete symbolic grammar learning devices are limited to discrete state spaces, RNNs are in principle suited for all sequence learning tasks because they have Turing capabilities (Siegelmann & Sontag, 1991). Typical RNN learning algorithms (Pearlmutter, 1995) 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. Hybrid HMM-RNN approaches (Bengio & Frasconi, 1995) might be able to combine the virtues of both methodologies, but to our knowledge have never been applied to the problem of precise event timing as discussed here.

Previous tasks already required the LSTM network to act upon events that occurred 50 discrete time steps ago, independently of what happened over the intervening 49 steps (see Chapter 3 and Hochreiter & Schmidhuber, 1997). Right before the critical moment, however,

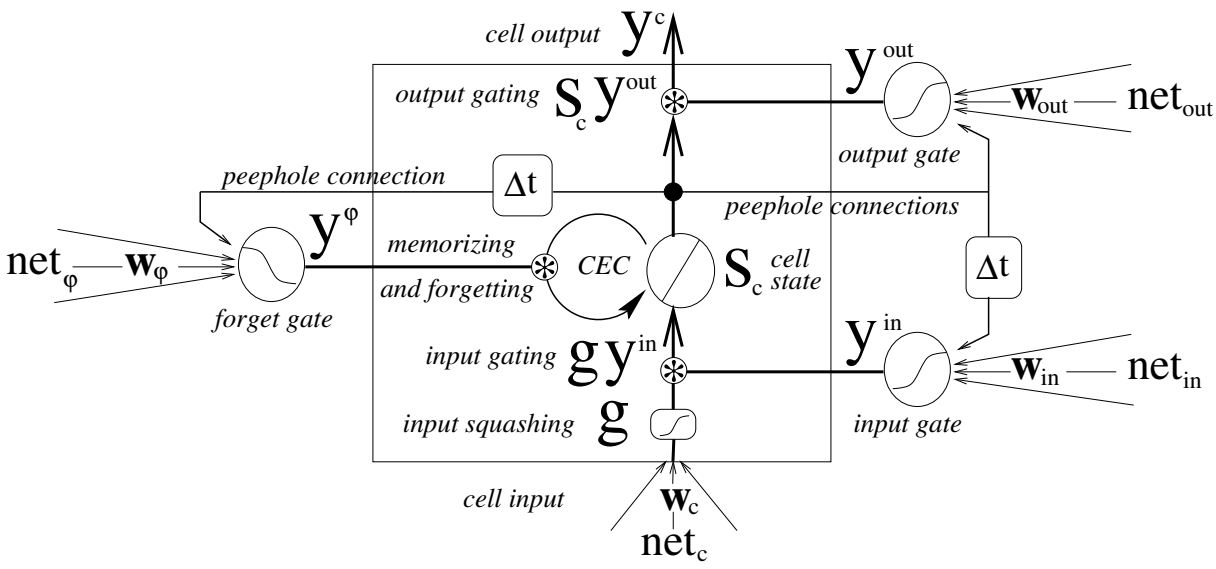

Figure 5.1: LSTM memory block with one cell; peephole connections connect s_c to the gates.

there was a helpful "marker" input informing the network that its next action would be crucial. Thus the network did not really have to learn to measure a time interval of 50 steps; it just had to learn to store relevant information for 50 steps, and use it once the marker was observed — something that is impossible for traditional RNNs but comparatively easy for LSTM.

But what if there are no such markers at all? What if the network itself has to learn to measure and internally represent the duration of task-specific intervals, or to *generate* sequences of patterns separated by exact intervals? Here we will study to what extent this is possible. The highly nonlinear tasks in the present chapter do not involve any time marker inputs; instead they require the network to time precisely and robustly across long time lags in continual input streams.

Before we describe our new timing experiments we will first identify a weakness in LSTM's connection scheme, and introduces peephole connections as a remedy (Section 5.2). Sections 5.3 and 5.4 describe the modified forward and backward pass for "peephole LSTM."

5.2 Extending LSTM with "Peephole Connections"

We are building on LSTM with forget gates (Chapter 3), simply called "LSTM" in what follows.

A limitation of LSTM. Each gate receives connections from the input units and the outputs of all cells. But there is no direct connection from the CEC it is supposed to control. All it can observe directly is the cell output, which is close to zero as long as the output gate is closed. The resulting lack of essential information may harm network performance, especially in case of the tasks we are going to study here.

Peephole connections. Our simple but very effective remedy is to add weighted "peephole" connections from the CEC to the gates of the same memory block (Figure 5.1). The gates learn to shield the CEC from unwanted inputs (forward pass) or unwanted error signals (backward pass). To keep the shield intact, during learning no error signals are propagated back from gates via peephole connections to the CEC (see backward pass, Section 5.4). Peephole connections are treated like regular connections to gates (e.g., from the input) except for update timing. For

conventional LSTM the only source of recurrent connections is the cell output y^c, so the order of updates within a layer is arbitrary. Peephole connections from within the cell, or recurrent connections from gates, however, require a refinement of LSTM's update scheme.

Updates for peephole LSTM. Each memory cell component should be updated based on the most recent activations or states of connected sources. In the simplest case this requires a two-phase update scheme; when recurrent connections from gates are present, the first phase must be further subdivided into three steps (a,b,c):

1. (a) Input gate activation y^{in},
   (b) forget gate activation y^{φ},
   (c) cell input and cell state s_c,
2. output gate activation y^{out} and cell output y^c.

Thus the output gate is updated after cell state s_c, seeing via its peephole connection the current value of $s_c(t)$ (already affected by forget gate and recent input), and possibly the current input and forget gate activations.

5.3 Forward Pass

Before specifying the equations for the LSTM model with peephole connections, we introduce a minor simplification unrelated to the central idea of this chapter. So far LSTM memory cells incorporated an input squashing function g *and* an output squashing function (called h in earlier LSTM publications). We remove the latter for lack of empirical evidence that it is really needed (in fact, the very first LSTM publication (Hochreiter & Schmidhuber, 1997) already omitted the output squashing function for some experiments).

Step 1a,1b. The input gate activation y^{in} and the forget gate activation y^{φ} are computed as:

$$net_{in_j}(t) = \sum_m w_{in_j m}\, y^m(t-1) + \sum_{v=1}^{S_j} w_{in_j c_j^v}\, s_{c_j^v}(t-1)\ , \quad y^{in_j}(t) = f_{in_j}(net_{in_j}(t))\ , \tag{5.1}$$

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m}\, y^m(t-1) + \sum_{v=1}^{S_j} w_{\varphi_j c_j^v}\, s_{c_j^v}(t-1)\ , \quad y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t))\ . \tag{5.2}$$

The peephole connections for the input gate and the forget gate are incorporated in equation 5.1 and 5.2 by including the CECs (containing the cell states) of memory block j as source units.

Step 1c. At $t = 0$, the state $s_c(t)$ of memory cell c is initialized to zero; subsequently ($t > 0$) it is calculated by adding the squashed, gated input to the state at the previous time step, $s_c(t-1)$, which is multiplied (gated) by the forget gate activation $y^{\varphi_j}(t)$:

$$\begin{aligned} net_{c_j^v}(t) &= \sum_m w_{c_j^v m}\, y^m(t-1)\ , \\ s_{c_j^v}(t) &= y^{\varphi_j}(t)\ s_{c_j^v}(t-1) + y^{in_j}(t)\ \ g(net_{c_j^v}(t))\ . \end{aligned} \tag{5.3}$$

Step 2. The output gate activation y^{out} is computed as:

$$\begin{aligned} net_{out_j}(t) &= \sum_m w_{out_j m}\, y^m(t-1) + \sum_{v=1}^{S_j} w_{out_j c_j^v}\, s_{c_j^v}(t)\ , \\ y^{out_j}(t) &= f_{out_j}(net_{out_j}(t))\ . \end{aligned} \tag{5.4}$$

Equation 5.4 includes the peephole connections for the output gate from the CECs of memory block j with the cell states $s_c(t)$, as updated in step 1c. The cell output y^c is computed as:

$$y^{c_j^v}(t) = y^{out_j}(t)\ s_{c_j^v}(t) \quad . \tag{5.5}$$

The equations for the output units k remain as specified in equations 2.9.

5.4 Gradient-Based Backward Pass

The revised update scheme for memory blocks allows for treating peephole connections like regular connections (see Sections 5.2 and 5.3), and so requires only minor changes to the backward pass (Chapter 3). We will present it below but not fully re-derive it. We will, however, point out the differences to the previous equations in Section 3.2.2. Appendix B gives pseudo-code for the entire algorithm.

In what follows we will present equations for LSTM with forget gates and peephole connections, but without output squashing. The sign $\stackrel{tr}{=}$ will indicate where we use error truncation.

During each step in the forward pass, no matter whether a target is given or not, we need to update the partial derivatives $\partial s_{c_j^v}/\partial w_{lm}$ and $\partial s_{c_j^v}/\partial w_{lc_j^{v'}}$ for weights to the cell ($l = c_j^v$), to the input gate ($l = in$), and to the forget gate ($l = \varphi$):

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}}\ y^{\varphi_j}(t) + g'(net_{c_j^v}(t))\ y^{in_j}(t)\ y^m(t-1)\ , \tag{5.6}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}}\ y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ f'_{in_j}(net_{in_j}(t))\ y^m(t-1)\ ; \tag{5.7a}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j c_j^{v'}}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j c_j^{v'}}}\ y^{\varphi_j}(t) + g(net_{c_j^v}(t))\ f'_{in_j}(net_{in_j}(t))\ s_{c_j^{v'}}(t-1)\ , \tag{5.7b}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}}\ y^{\varphi_j}(t) + s_{c_j^v}(t-1)\ f'_{\varphi_j}(net_{\varphi_j}(t))\ y^m(t-1)\ ; \tag{5.8a}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j c_j^{v'}}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j c_j^{v'}}}\ y^{\varphi_j}(t) + s_{c_j^v}(t-1)\ f'_{\varphi_j}(net_{\varphi_j}(t))\ s_{c_j^{v'}}(t-1)\ , \tag{5.8b}$$

with $\partial s_{c_j^v}(0)/\partial w_{lm} = \partial s_{c_j^v}(0)/\partial w_{lc_j^{v'}} = 0 \quad \text{for } l \in \{in, \varphi, c_j^v\}$. Equation 5.7b and 5.8b are for the peephole connection weights.

Following previous notation, we minimize the objective function E by gradient descent (subject to error truncation), changing the weights w_{lm} (from unit m to unit l) by an amount Δw_{lm} given by the learning rate α times the negative gradient of E. For the output units we obtain the standard back-propagation weight changes:

$$\Delta w_{km}(t) = \alpha\ \delta_k(t)\ y^m(t-1) \quad , \quad \delta_k(t) = \frac{\partial E(t)}{\partial net_k(t)} \ . \tag{5.9}$$

Here we use the customary squared error objective function based on targets t^k, yielding:

$$\delta_k(t) = f'_k(net_k(t))\ e_k(t)\ , \tag{5.10}$$

where $e_k(t) := t^k(t) - y^k(t)$ is the externally injected error. The weight changes for connections to the output gate (of the j-th memory block) from the source units (as specified by the network topology) $w_{out_j m}$ and for the peephole connections $w_{out_j c_j^v}$ are:

$$\Delta w_{out_j m}(t) = \alpha\ \delta_{out_j}(t)\ y^m(t)\ , \quad \Delta w_{out_j c_j^v}(t) = \alpha\ \delta_{out_j}(t)\ s_{c_j^v}(t)\,, \tag{5.11a}$$

$$\delta_{out_j}(t) \stackrel{tr}{=} f'_{out_j}(net_{out_j}(t)) \left(\sum_{v=1}^{S_j} s_{c_j^v}(t) \sum_k w_{kc_j^v}\ \delta_k(t) \right)\ . \tag{5.11b}$$

Output squashing (removed here) would require the incorporation of the derivative of the output squashing function in (5.11b). To calculate weight changes Δw_{lm} and $\Delta w_{lc_j^{v'}}$ (peephole connection weights) for connections to the cell ($l = c_j^v$), the input gate ($l = in$), and the forget gate ($l = \varphi$) we use the partials from Equations 5.6, 5.7b, and 5.8b:

$$\Delta w_{c_j^v m}(t) = \alpha\ e_{s_{c_j^v}}(t)\ \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \tag{5.12}$$

$$\Delta w_{in_j m}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t)\ \frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}}\ , \quad \Delta w_{in_j c_j^{v'}}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t)\ \frac{\partial s_{c_j^v}(t)}{\partial w_{in_j c_j^{v'}}} \tag{5.13}$$

$$\Delta w_{\varphi_j m}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t)\ \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}}\ , \quad \Delta w_{\varphi_j c_j^{v'}}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t)\ \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j c_j^{v'}}} \tag{5.14}$$

where the internal state error $e_{s_{c_j^v}}$ is separately calculated for each memory cell:

$$e_{s_{c_j^v}}(t) \stackrel{tr}{=} y^{out_j}(t) \left(\sum_k w_{kc_j^v}\ \delta_k(t) \right)\ . \tag{5.15}$$

Like traditional LSTM LSTM with forget gates and peephole connections is still local in space *and* time. The increase in complexity due to peephole connections is small: 3 weights per cell.

5.5 Experiments

We study LSTM's performance on three tasks that require the precise measurement or generation of delays. We compare conventional to peephole LSTM, analyze the solutions, or explain why none was found.

Measuring spike delays (MSD). See Section 5.5.2. The goal is to classify input sequences consisting of sharp spikes. The class depends on the interval between spikes. We consider two versions of the task: continual (MSD) and non-continual (NMSD). NMSD sequences stop after the second spike, whereas MSD sequences are continual spike trains. Both NMSD and MSD require the network to measure intervals between spikes; MSD also requires the production of stable results in presence of continually streaming inputs, without any external reset of the network's state. Can LSTM learn the difference between almost identical pattern sequences that differ only by a small lengthening of the interval (e.g., from n to $n+1$ steps) between input spikes? How does the difficulty of this problem depend on n?

Generating timed spikes (GTS). See Section 5.5.3. The GTS task can be obtained from the MSD task by exchanging inputs and targets. It requires the production of continual spike trains, where the interval between spikes must reflect the magnitude of an input signal that may change after every spike.

GTS is a special case of periodic function generation (PFG, see below). In contrast to previously studied PFG tasks (Williams & Zipser, 1989; Doya & Yoshizawa, 1989; Tsung & Cottrell, 1995), GTS is highly nonlinear and involves long time lags between significant output changes, which cannot be learned by conventional RNNs. Previous work also did not focus on stability issues. Here, by contrast, we demand that the generation be stable for 1000 successive spikes. We systematically investigate the effect of minimal time lag on task difficulty.

Additional periodic function generation tasks (PFG). See Section 5.5.4. We study the problem of generating periodic functions other than the spike trains above. The classic examples are smoothly oscillating outputs such as sine waves, which are learnable by fully connected teacher-forced RNNs whose units are all output units with teacher-defined activations (Williams & Zipser, 1989). An alternative approach trains an RNN to predict the next input; after training outputs are fed back directly to the input so as to generate the waveform (Doya & Yoshizawa, 1989; Tsung & Cottrell, 1995; Weiss, 1999; Townley et al., 1999).

Here we focus on more difficult, highly nonlinear, triangular and rectangular waveforms, the latter featuring long time lags between significant output changes. Again, traditional RNNs cannot learn tasks involving long time lags (Hochreiter, 1991; Bengio et al., 1994), and previous work did not focus on stability issues. By contrast, we demand that the generation be stable for 1000 successive periods of the waveform.

5.5.1 Network Topology and Experimental Parameters

We found that comparatively small LSTM nets can already solve the tasks above. A single input unit (used only for tasks where there is input) is fully connected to the hidden layer consisting of a single memory block with one cell. The cell output is connected to the cell input, to all three gates, and to a single output unit (Figure 5.2). All gates, the cell itself, and the output unit are connected to a bias unit (a unit with constant activation one) as well. The bias weights to input gate, forget gate, and output gate are initialized to 0.0, -2.0 and $+2.0$, respectively. (Although not critical, these values have been found empirically to work well; we use them for all our experiments.) All other weights are initialized to uniform random values in the range $[-0.1, 0.1]$. In addition to the three peephole connections there are 14 adjustable weights: 9 "unit-to-unit" connections and 5 bias connections. The cell's input squashing function g is the identity function. The squashing function of the output unit is a logistic sigmoid with range $[0, 1]$ for MSD and GTS (except where explicitly stated otherwise), and the identity function for PFG. (A sigmoid function would work as well, but we focus on the simplest system that can solve the task.)

Our networks process continual streams of inputs and targets; only at the beginning of a stream are they reset. They must learn to always predict the target $t_k(t)$, producing a stream of output values (predictions) $y_k(t)$. A prediction is considered correct if the absolute output error $|e_k(t)| = |t^k(t) - y^k(t)|$ is below 0.49 for binary targets (MSD, NMSD and GTS tasks), below 0.3 otherwise (PFG tasks). Streams are stopped as soon as the network makes an incorrect prediction, or after a given maximal number of successive periods (spikes): 100 during training, 1000 during testing.

Learning and testing alternate: after each training stream, we freeze the weights and generate

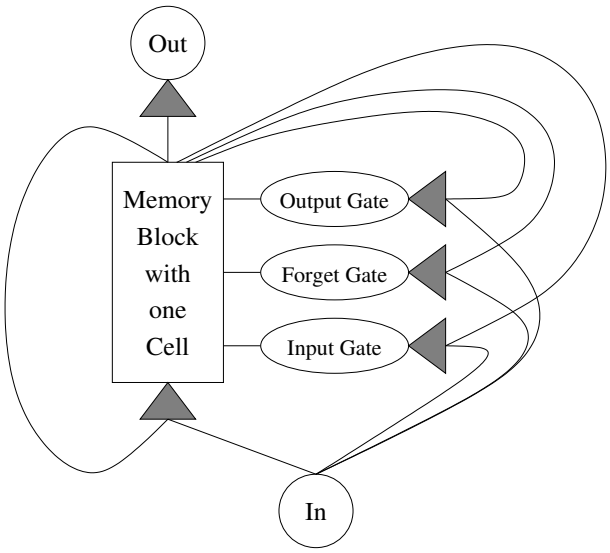

Figure 5.2: Three-layer LSTM topology with one input and one output unit. Recurrence is limited to the hidden layer, which consists of a single LSTM memory block with a single cell. All 9 "unit-to-unit" connections are shown, but bias and peephole connections are not.

a test stream. Our performance measure is the achieved test stream size: 1000 successive periods are deemed a "perfect" solution. Training is stopped once a task is learned or after a maximal number of 10^7 training streams (10^8 for the MSD and NMSD tasks). Weight changes are made after each target presentation. The learning rate α is set to 10^{-5}; we use the momentum algorithm (Plaut, Nowlan, & Hinton, 1986) with momentum parameter 0.999 for the GTS task, 0.99 for the PFG and NMSD task, and 0.9999 for the MSD task. We roughly optimized the momentum parameter by trying out different orders of magnitude.

For tasks GTS and MSD, the stochastic input streams are generated online. A perfect solution correctly processes 10 test streams, to make sure the network provides stable performance independent of the stream beginning, which we found to be critical. All results are averages over 10 independently trained networks.

5.5.2 Measuring Spike Delays (MSD)

The network input is a spike train, represented by a series of ones and zeros, where each "one" indicates a spike. Spikes occur at times $T(n)$ set $F + I(n)$ steps apart, where F is the minimum interval between spikes, and $I(n)$ is an integer offset, randomly reset for each spike:

$$T(0) = F + I(0) \ , \ T(n) = T(n-1) + F + I(n) \ (n \in \mathbb{N}) \ .$$

The target given at times $t = T(n)$ is the delay $I(n)$. (Learning to measure the total interval $F + I(n)$ — that is, adding the constant F to the output — is no harder.) A perfect solution correctly processes all possible input test streams. For the non-continual version of the task (NMSD) a stream consists of a single period (spike).

MSD Results. Table 5.1 reports results for NMSD with $I(n) \in \{0, 1\}$ for various minimum spike intervals F. The results suggest that the difficulty of the task (measured as the average number of training streams necessary to solve it) increases drastically with F (see Figure 5.3). A qualitative explanation is that longer intervals necessitate finer tuning of the weights, which

T	F	$I(n) \in$	LSTM		Peephole LSTM	
			% Sol.	Train. [10^3]	% Sol.	Train. [10^3]
NMSD	10	$\{0,1\}$	100	160 ± 14	100	125 ± 14
	20	$\{0,1\}$	100	732 ± 97	100	763 ± 103
	30	$\{0,1\}$	100	17521 ± 2200	80	12885 ± 2091
	40	$\{0,1\}$	20	37533 ± 4558	70	25686 ± 2754
	50	$\{0,1\}$	0	—	10	32485
MSD	10	$\{0,1\}$	10	8850	20	27453 ± 11750
	10	$\{0,1,2\}$	20	29257 ± 13758	60	9791 ± 2660

Table 5.1: Results comparing conventional and peephole LSTM on the NMSD and MSD tasks. Columns show the task T, the minimum spike interval F, the set of delays $I(n)$, the percentage of perfect solutions found, and the mean and standard derivation of the number of training streams required.

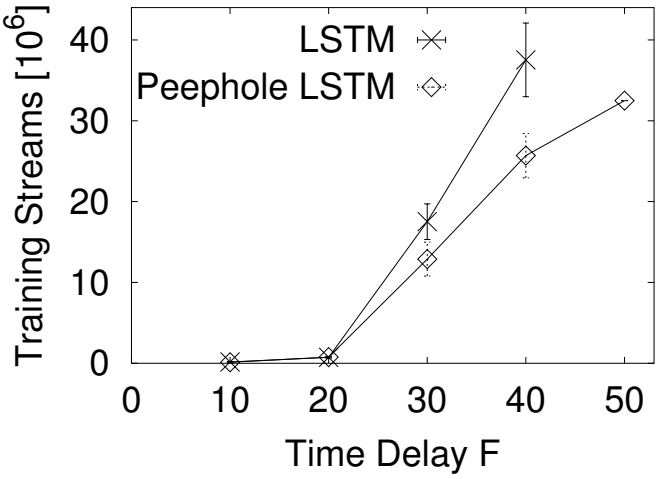

Figure 5.3: Average number of training streams required for the NMSD task with $I(n) \in \{0,1\}$, plotted against the minimum spike interval F.

requires more training. Peephole LSTM outperforms LSTM. The continual MSD task for $F = 10$ with $I(n) \in \{0,1\}$ or $I(n) \in \{0,1,2\}$, is solved with or without peephole connections (Table 5.1).

In the next experiment we evaluate the influence of the range of $I(n)$, using the identity function instead of the logistic sigmoid as output squashing function. We let $I(n)$ range over $\{0,i\}$ or $\{0,..,i\}$ for all $i \in \{1,..,10\}$. Results are reported in Table 5.2 for NMSD with $F = 10$. The training duration depends on the size of the set from which $I(n)$ is drawn, and on the maximum distance (MD) between elements in the set. A larger MD leads to a better separation of patterns, thus facilitating recognition. To confirm this, we ran the NMSD task with $F = 10$ and $I(n) \in \{0,i\}$ with $i \in \{2,..,10\}$ (size 2, MD i), as shown in the bottom half of Table 5.2. As expected, training time decreases with increasing MD. A larger set of possible delays should make the task harder. Surprisingly, for $I(n) \in \{0,..,i\}$ (size $i+1$, MD i) with i ranging from 1 to 5 the task appears to become easier (due to the simultaneous increase of MD) before the

$I(n) \in$	LSTM		Peephole LSTM	
	% Sol.	Training Str. [10^3]	% Sol.	Training Str. [10^3]
$\{0,1\}$	100	48 ± 12	100	46 ± 14
$\{0,1,2\}$	100	25 ± 4	100	10.3 ± 3.3
$\{0,..,3\}$	100	12.3 ± 2.4	100	7.4 ± 2.2
$\{0,..,4\}$	100	8.5 ± 1.3	100	3.6 ± 0.4
$\{0,..,5\}$	100	4.5 ± 0.4	100	6.0 ± 1.4
$\{0,..,6\}$	100	6.1 ± 1.0	100	7.1 ± 2.8
$\{0,..,7\}$	100	8.5 ± 2.9	70	15 ± 6.5
$\{0,..,8\}$	100	14.1 ± 4.2	50	22 ± 9
$\{0,..,9\}$	90	39 ± 28	50	33 ± 17
$\{0,..,10\}$	60	23 ± 5	20	395 ± 167
$\{0,2\}$	100	33 ± 8	100	18 ± 5
$\{0,3\}$	100	12.5 ± 4.2	100	23 ± 6
$\{0,4\}$	100	12.1 ± 2.8	100	13.7 ± 2.7
$\{0,5\}$	100	8.5 ± 2.3	100	10.4 ± 2.0
$\{0,6\}$	100	7.7 ± 1.5	100	12.7 ± 3.1
$\{0,7\}$	100	7.7 ± 1.5	100	14.5 ± 6.0
$\{0,8\}$	100	7.5 ± 2.0	100	6.3 ± 1.3
$\{0,9\}$	100	5.8 ± 1.6	100	7.5 ± 1.6
$\{0,10\}$	100	5.6 ± 0.9	100	6.7 ± 1.7

Table 5.2: The percentage of perfect solutions found, and the mean and standard derivation of the number of training streams required, for conventional versus peephole LSTM on the NMSD task with $F = 10$ and various choices for the set of delays $I(n)$.

difficulty increases rapidly for larger i. Thus the task's difficulty does not grow linearly with the number of possible delays, corresponding to values (states) inside a cell the network must learn to distinguish. Instead we observe that LSTM fares best at distinguishing 6 or 7 different delays. One is tempted to draw a connection to the "magic number" of 7 ± 2 items that an average human can store in Short Term Memory (STM) (Miller, 1956), but such a link seems rather far-fetched to us.

We also observe that the results for $I(n) \in \{0,1\}$ are better than those obtained with a sigmoid function (compare Table 5.1). Fluctuations in the stochastic input can cause temporary saturation of sigmoid units; the resulting tiny derivatives for the backward pass will slow down learning (LeCun, Bottou, Orr, & Müller, 1998).

MSD Analysis. LSTM learned to measure time in two principled ways. The first is to slightly increase the cell contents s_c at each time step, so that the elapsed time can be read off the value of s_c. This kind of solution is shown on the left-hand side of Figure 5.4. (The state reset performed by the forget gate is essential only for continual online prediction over many periods.) The second way is to establish internal oscillators and derive the elapsed time from their phases (right-hand side of Figure 5.4). Both kinds of solutions can be learned with or without peephole connections, as it is never necessary here to close the output gate for more than one time step (see bottom row of Figure 5.4).

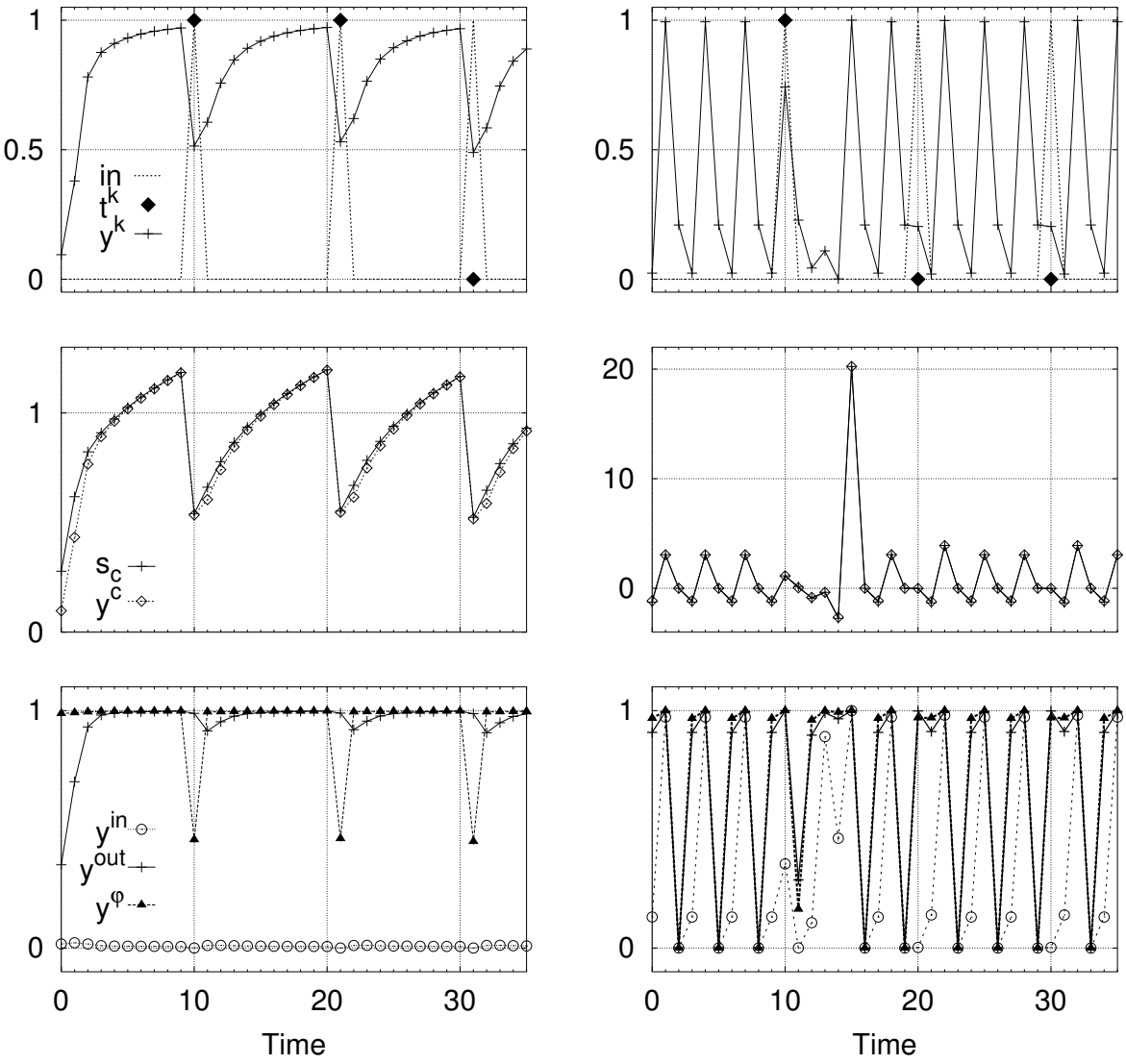

Figure 5.4: **Two ways to time.** Test run with trained LSTM networks for the MSD task with $F = 10$ and $I(n) \in \{0, 1\}$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

Why may the output gate be left open? Targets occur rarely, hence the network output can be ignored most of the time. Since there is only one memory block, mutual perturbation of blocks is not possible. This type of reasoning is invalid though for more complex measuring tasks involving larger nets or more frequent targets. Figure 5.5 shows the behavior of LSTM in such a regime. With peephole LSTM the output gate opens only when a target is provided, whereas conventional LSTM does not learn this behavior. Note that in some cases these "cleaner" solutions with peephole connections took longer to be learned (compare Tables 5.1 and 5.2, because they require more complex behavior.

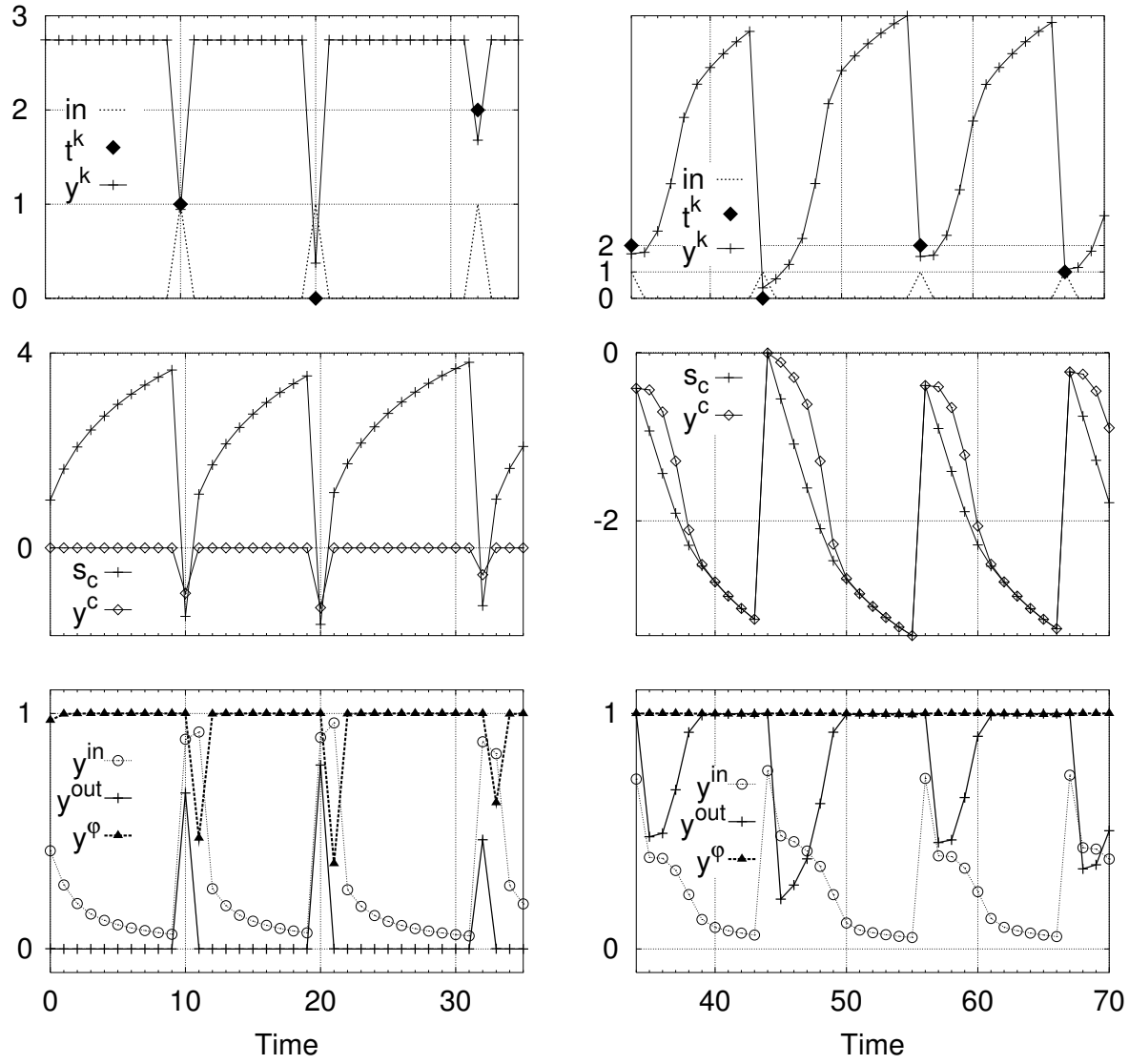

Figure 5.5: Behavior of peephole LSTM (left) versus LSTM (right) for the MSD task with $F = 10$ and $I(n) \in \{0, 1, 2\}$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

5.5.3 Generating Timed Spikes (GTS).

The GTS task reverses the roles of inputs and targets of the MSD task: the spike train $T(n)$, defined as for the MSD task, now is the network's target, while the delay $I(n)$ is provided as input.

GTS Results. The GTS task could *not* be learned by networks without peephole connections; thus we report results with peephole LSTM only. Results with various minimum spike intervals F (Figure 5.6) suggest that the required training time increases dramatically with F, as with the NMSD task (Section 5.5.2). The network output during a successful test run for the GTS task with $F = 10$ is shown on the top left of Figure 5.7. Peephole LSTM also solves the task for $F = 10$ and $I(n) \in \{0, 1\}$ or $\{0, 1, 2\}$, as shown in Figure 5.6 (left).

F	$I(n) \in$	Peephole LSTM	
		% Sol.	Train. $[10^3]$
10	$\{0\}$	100	41 ± 4
20	$\{0\}$	100	67 ± 8
30	$\{0\}$	80	845 ± 82
40	$\{0\}$	100	1152 ± 101
50	$\{0\}$	100	2538 ± 343
10	$\{0,1\}$	50	1647 ± 46
10	$\{0,1,2\}$	30	954 ± 393

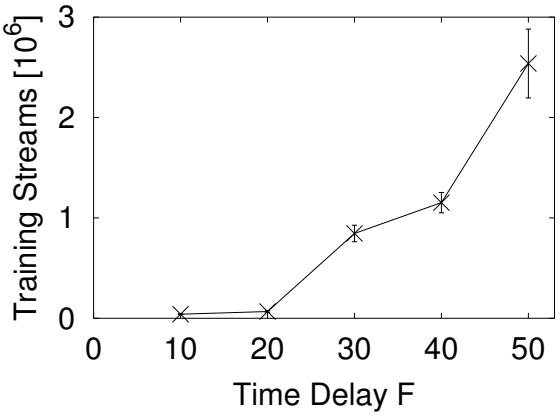

Figure 5.6: Results for the GTS task. Table (left) shows the minimum spike interval F, the set of delays $I(n)$, the percentage of perfect solutions found, and the mean and standard derivation of the number of training streams required. Graph (right) plots the number of training streams against the minimum spike interval F, for $I(n) \in \{0\}$.

GTS Analysis. Figure 5.7 shows test runs with trained networks for the GTS task. The output gates open only at the onset of a spike and close again immediately afterwards. Hence, during a spike, the output of the cell equals its state (middle row of Figure 5.7). The opening of the output gate is triggered by the cell state s_c: it starts to open once the input from the peephole connection outweighs a negative bias. The opening self-reinforces via a connection from the cell output, which produces the high nonlinearity necessary for generating the spike. This process is terminated by the closing of the forget gate, triggered by the cell output spike. Simultaneously the input gate closes, so that s_c is reset.

In the particular solution shown on the right-hand side of Figure 5.7 for $F = 50$, the role of the forget gate in this process is taken over by a negative self-recurrent connection of the cell in conjunction with a simultaneous opening of the other two gates. We tentatively removed the forget gate (by pinning its activation to 1.0) without changing the weights learned *with* the forget gate's help. The network then quickly learned a perfect solution. Learning from scratch *without* forget gate, however, never yields a solution! The forget gate is essential during the learning phase, where it prevents the accumulation of irrelevant errors.

The exact timing of a spike is determined by the growth of s_c, which is tuned through connections to input gate, forget gate, and the cell itself. To solve GTS for $I(n) \in \{0,1\}$ or $I(n) \in \{0,1,2\}$, the network essentially translates the input into a scaling factor for the growth of s_c (Figure 5.8).

5.5.4 Periodic Function Generation (PFG)

We now train LSTM to generate real-valued periodic functions, as opposed to the spike trains of the GTS task. At each discrete time step we provide a real-valued target, sampled with frequency F from a target function $f(t)$. No input is given to the network.

The task's degree of difficulty is influenced by the shape of f and the sampling frequency F. The former can be partially characterized by the absolute maximal values of its first and second derivatives, $\max|f'|$ and $\max|f''|$. Since we work in discrete time, and with non-differentiable

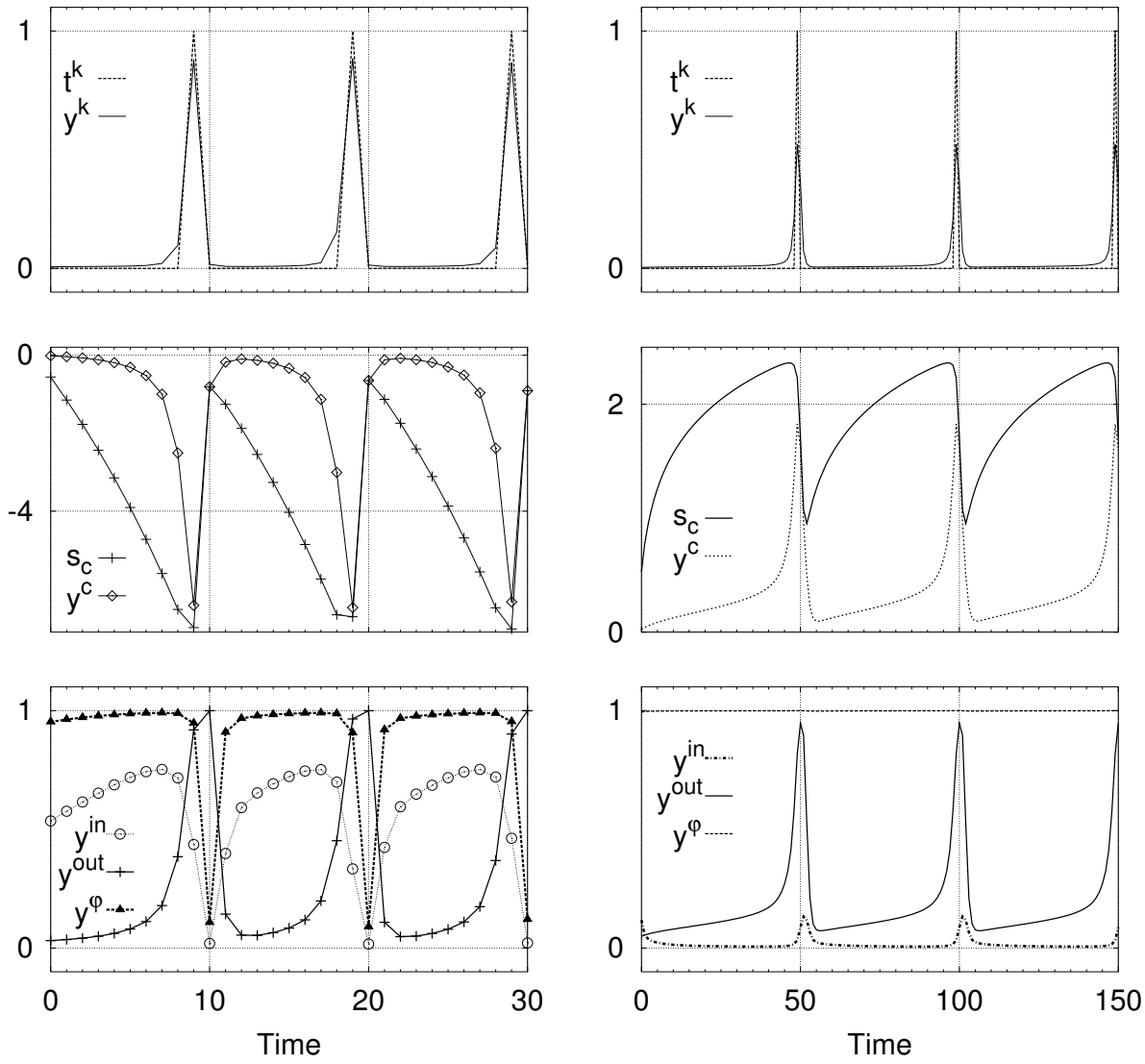

Figure 5.7: Test run of a trained peephole LSTM network for the GTS task with $I(n) \in \{0\}$, and a minimum spike interval of $F = 10$ (left) vs. $F = 50$ (right). Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

step functions, we define:

$$f'(t) := f(t+1) - f(t) \ , \quad \max|f'| \equiv \max_t |f'(t)| \ , \quad \max|f''| \equiv \max_t |f'(t+1) - f(t)'| \ .$$

Generally speaking, the larger these values, the harder the task. F determines the number of distinguishable internal states required to represent the periodic function in internal state space. The larger F, the harder the task. We generate sine waves $f_{\cos}$, triangular functions f_{tri}, and rectangular functions f_{rect}, all ranging between 0.0 and 1.0, each sampled with two frequencies,

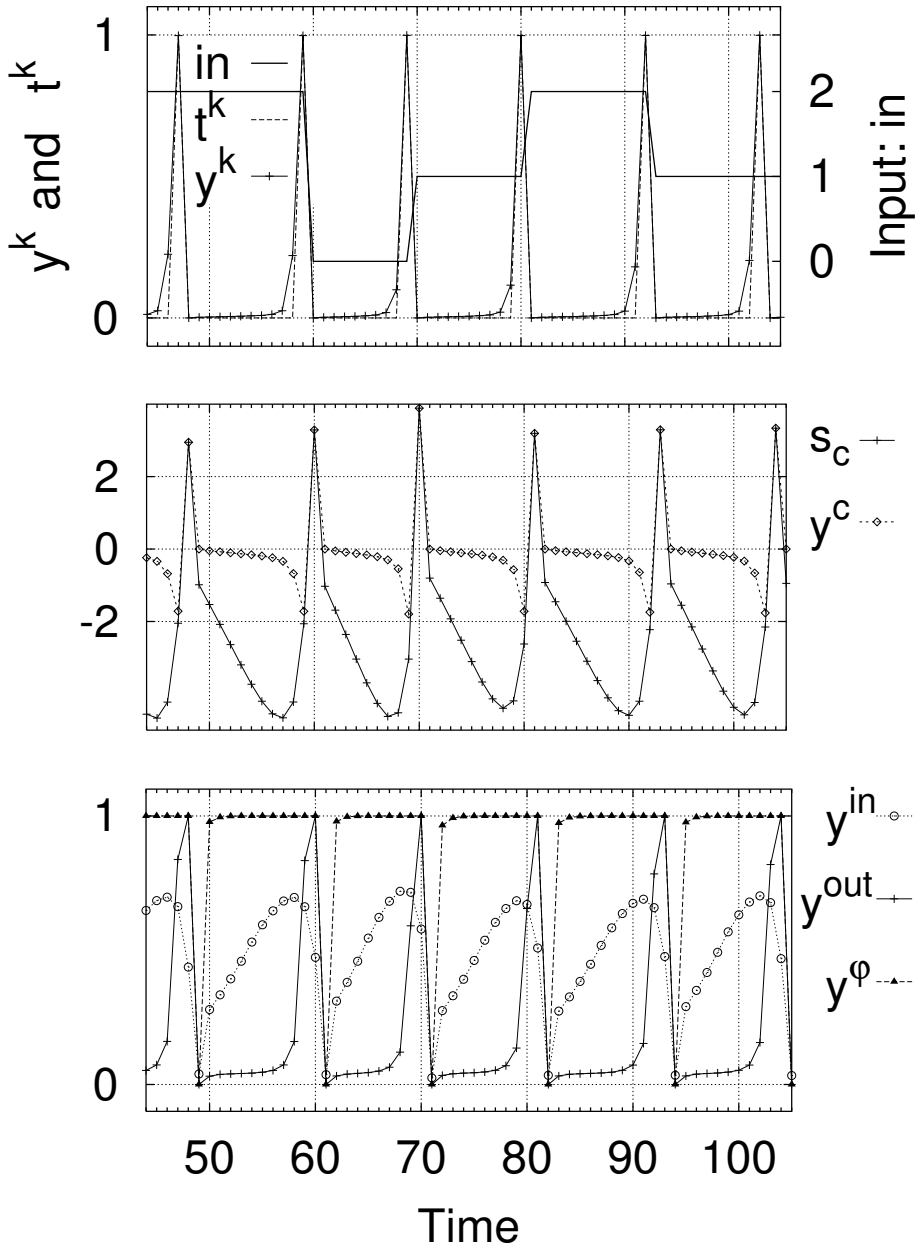

Figure 5.8: Test run of a trained peephole LSTM network for the GTS task with $F = 10$ and $I(n) \in \{0, 1, 2\}$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

$F = 10$ and $F = 25$:

$$f_{\cos}(t) \equiv \frac{1}{2}\left(1 - \cos\left(\frac{2\pi\ t}{F}\right)\right) \qquad \Rightarrow\ \max|f'_{\cos}| = \max|f''_{\cos}| = \pi/F\ ,$$

$$f_{\mathrm{tri}}(t) \equiv \begin{cases} \frac{2\ (t \bmod F)}{F} & \text{if } (t \bmod F) > \frac{F}{2} \\ 2 - \frac{2\ (t \bmod F)}{F} & \text{otherwise} \end{cases} \qquad \Rightarrow\ \max|f'_{\mathrm{tri}}| = 2/F\ ,\ \max|f''_{\mathrm{tri}}| = 4/F\ ,$$

$$f_{\mathrm{rect}}(t) \equiv \begin{cases} 1 & \text{if } (t \bmod T) > \frac{F}{2} \\ 0 & \text{otherwise} \end{cases} \qquad \Rightarrow\ \max|f'_{\mathrm{rect}}| = \max|f''_{\mathrm{rect}}| = 1\ .$$

tgt. fn.	F	LSTM			Peephole LSTM		
		% Sol.	Training Str. [10^3]	$\sqrt{MSE}$	% Sol.	Training Str. [10^3]	$\sqrt{MSE}$
$f_{\cos}$	10	90	2477 ± 341	0.13 ± 0.033	100	145 ± 32	0.18 ± 0.016
	25	0	> 10000	—	60	149 ± 7	0.17 ± 0.019
f_{tri}	10	0	> 10000	—	100	869 ± 204	0.13 ± 0.014
	25	0	> 10000	—	50	4063 ± 303	0.13 ± 0.024
f_{rect}	10	0	> 10000	—	80	1107 ± 97	0.12 ± 0.014
	25	0	> 10000	—	20	748 ± 278	0.12 ± 0.012

Table 5.3: Results for the PFG task, showing target function f, sampling frequency F, the percentage of perfect solutions found, and the mean and standard derivation of the number of training streams required, as well as of the root mean squared error $\sqrt{MSE}$ for the final test run.

PFG Results. Our experimental results for the PFG task are summarized in Table 5.3. Peephole LSTM found perfect, stable solutions for all target functions (Figure 5.9). LSTM without peephole connections could solve only $f_{\cos}$ with $F = 10$, requiring many more training streams. Without forget gates, LSTM never learned to predict the waveform for more than two successive periods.

The duration of training roughly reflected our criteria for task difficulty. We did not try to achieve maximal accuracy for each task: training was stopped once the "perfect solution" criteria were fulfilled. Accuracy can be improved by decreasing the tolerated maximum output error e_k^{max} during training, albeit at a significant increase in training duration. Decreasing e_k^{max} by one half (to 0.15) for $f_{\cos}$ with $F = 25$ also reduces the average $\sqrt{MSE}$ of solutions by about one half, from 0.17 ± 0.019 down to 0.086 ± 0.002. Perfect solutions were learned in all cases, but only after $(2704 \pm 49) \cdot 10^3$ training streams, as opposed to $(149 \pm 7) \cdot 10^3$ training streams (yielding 60% solutions) before.

PFG Analysis. For the PFG task, the networks do not have any external input, so updates depend on the internal cell states only. Hence, in a stable solution for a periodic target function $t_k(t)$ the cell states s_c also have to follow some periodic trajectory $s(t)$ phase-locked to $t_k(t)$. Since the cell output is the only time-varying input to gates and output units, it must simultaneously minimize the error at the output units and provide adequate input to the gates. An example of how these two requirement can be combined in one solution is shown in Figure 5.10 for $f_{\cos}$ with $F = 10$. This task can be solved with or without peephole connections because the output gate never needs to be closed completely, so that all gates can base their control on the cell output.

Why did LSTM networks without peephole connections never learn the target function $f_{\cos}$ for $F = 25$, although they did learn it for $F = 10$? The output gate is part of an uncontrolled feedback loop: its activation directly determines its own input (here: its *only* input, except for the bias) via the connection to the cell output — but no errors are propagated back on this connection. The same is true for the other gates, except that output gating can block their (thus incomplete) feedback loop. This makes an adaptive LSTM memory block without peephole connections more difficult to tune. Additional support for this reasoning stems from the fact that networks with peephole connections learn $f_{\cos}$ with $F = 10$ much faster (see Table 5.3). The peephole weights of solutions are typically of the same magnitude as the weights of

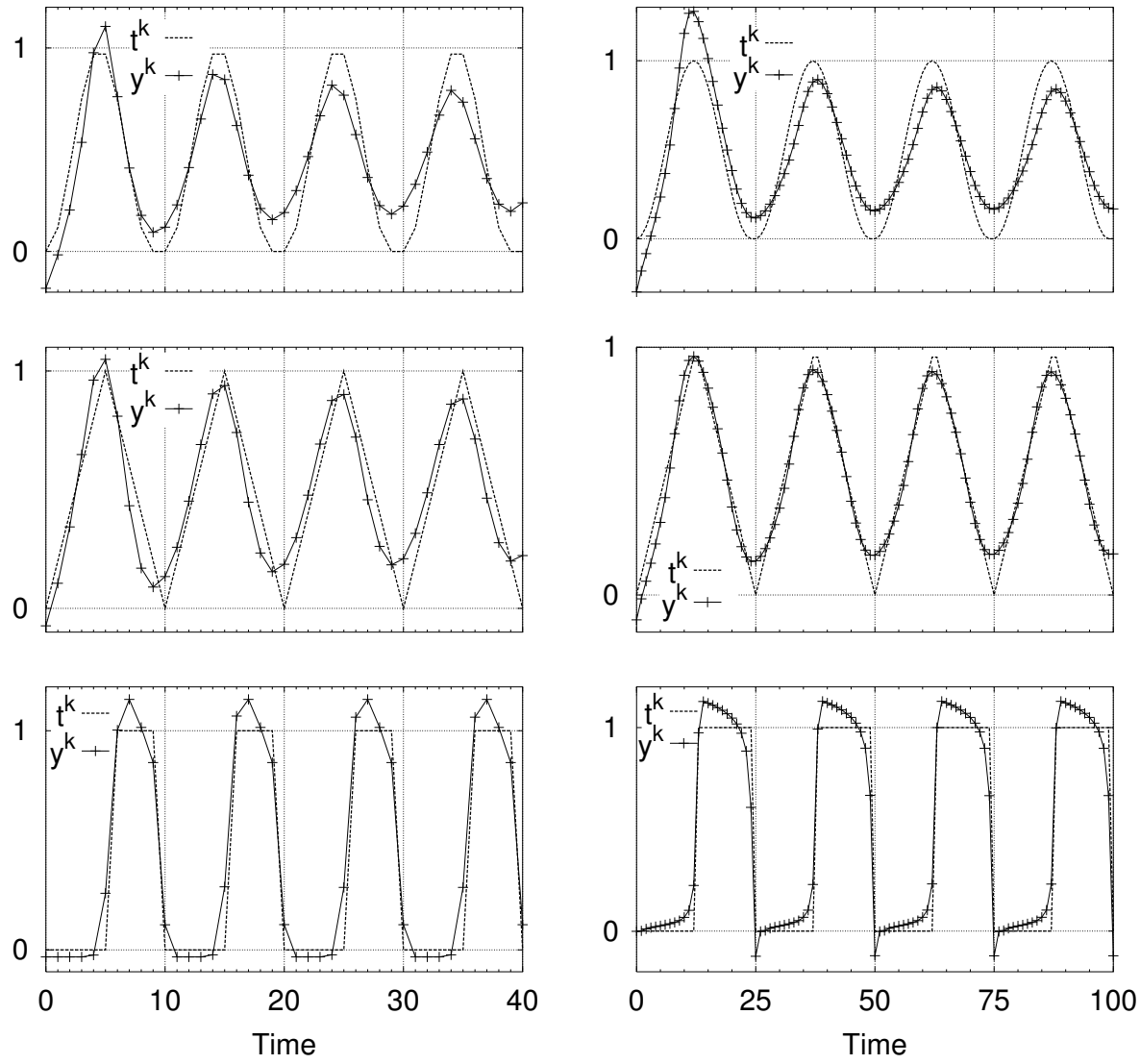

Figure 5.9: Target values t_k and network output y_k during test runs of trained peephole LSTM networks on the PFG task for the periodic functions $f_{\cos}$ (top), f_{tri} (middle), and f_{rect} (bottom), with periods $F = 10$ (left) and $F = 25$ (right).

connections from cell output to gates, which shows that they are indeed used even though they are not mandatory for this task.

The target functions f_{tri} and f_{rect} *required* peephole connections for both values of F. Figure 5.11 shows typical network solutions for the f_{rect} target function. The cell output y^c equals the cell state s_c in the second half of each period (when $f_{\text{rect}} = 1$) and is zero in the first half, because the output gate closes the cell (triggered by s_c, which is accessed via the peephole connections). The timing information is read off s_c, as explained in Section 5.5.2. Furthermore, the two states of the f_{rect} function are distinguished: s_c is counted up when $f_{\text{rect}} = 0$ and counted down again when $f_{\text{rect}} = 1$. This is achieved through a negative connection from the cell output to the the cell input, feeding negative input into the cell only when the output gate

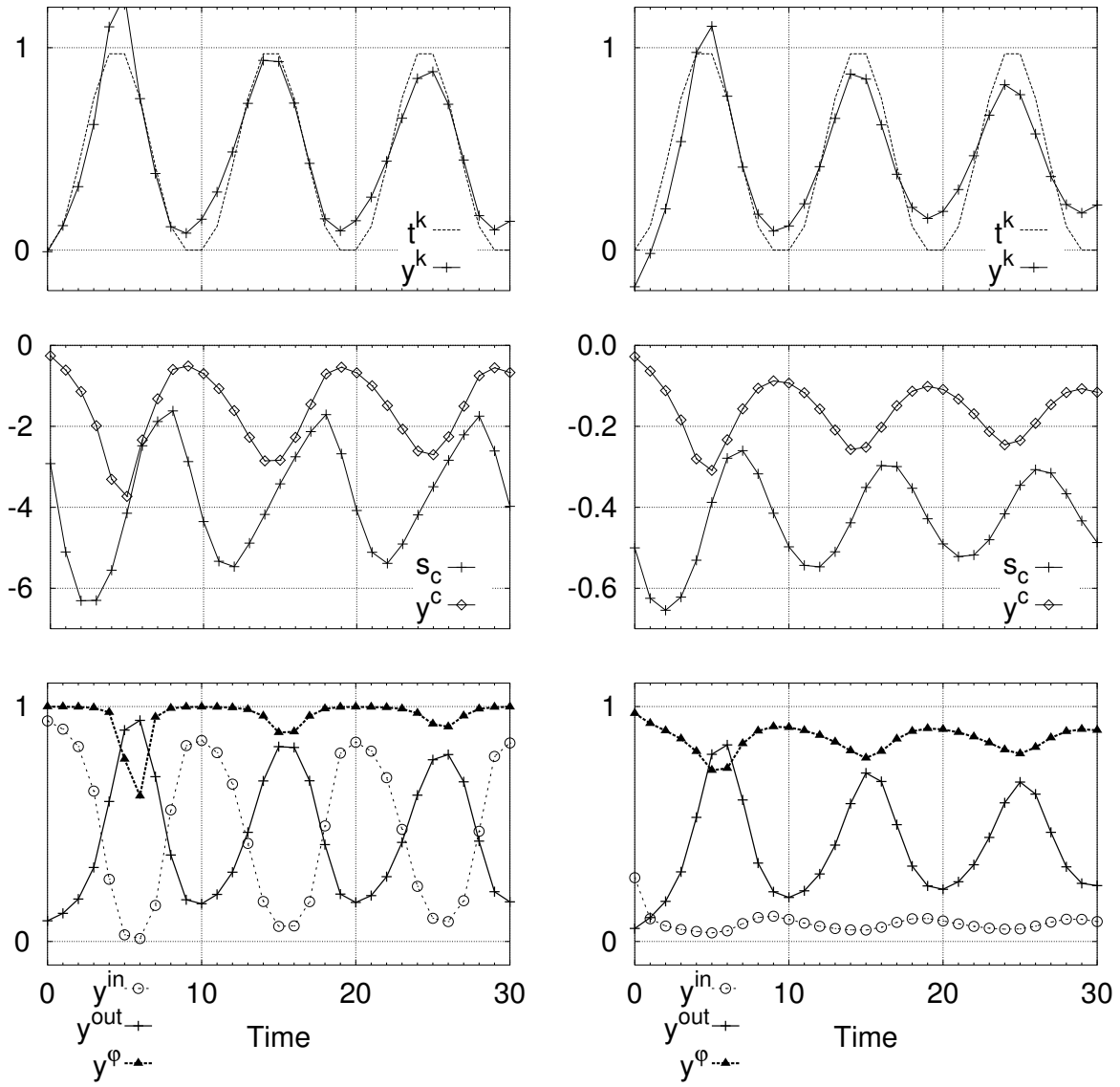

Figure 5.10: Test runs of a trained LSTM network with (right) vs. without (left) peephole connections on the $f_{\cos}$ PFG task with $F = 10$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

is open; otherwise the input is dominated by the positive bias connection. Networks without peephole connections cannot use this mechanism, and did not find any alternative solution. Throughout all experiments peephole connections were necessary to trigger the opening of gates while the output gate was closed, by granting unrestricted access to the timer implemented by the CEC. The gates learned to combine this information with their bias so as to open on reaching a certain trigger threshold.

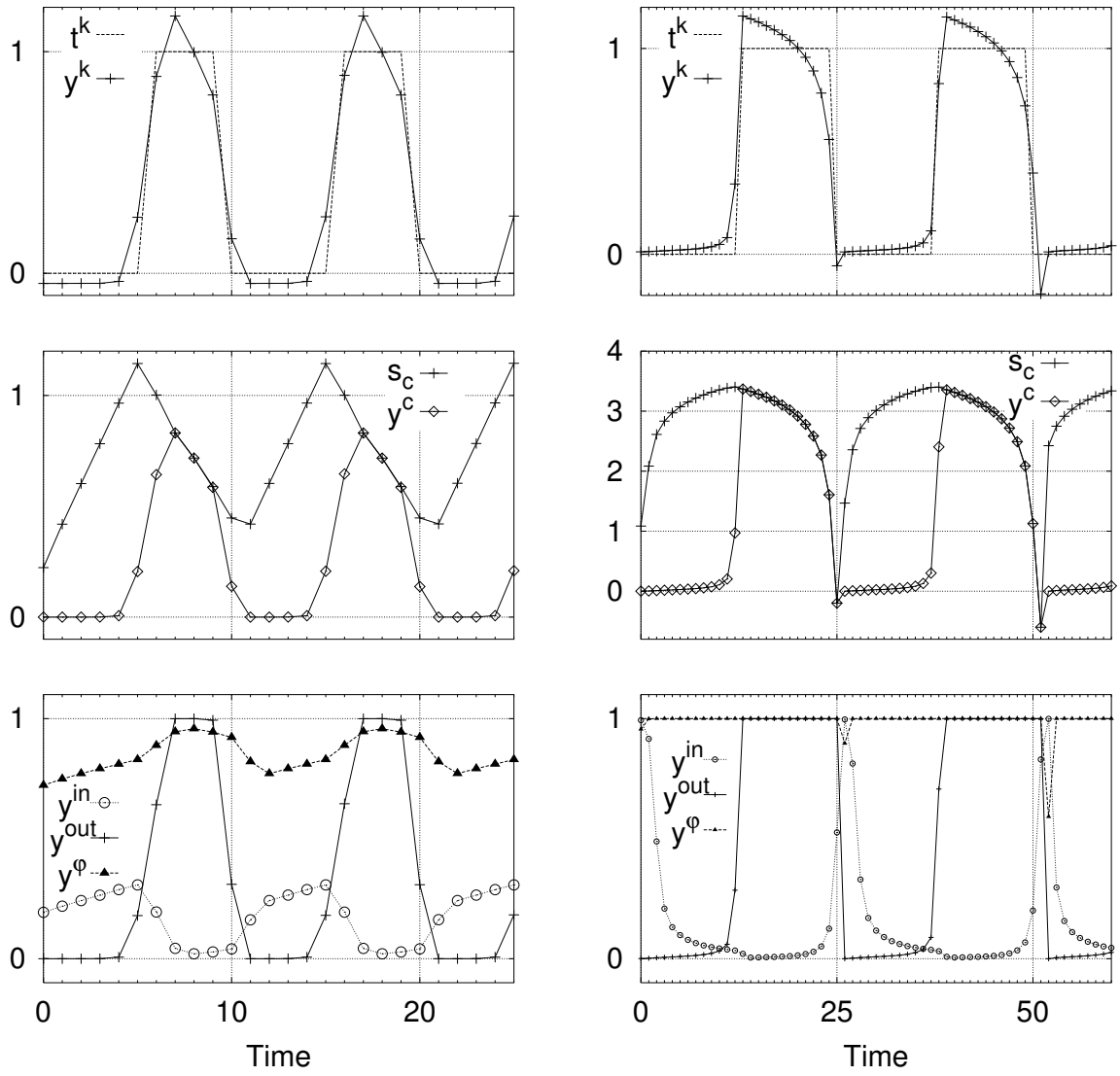

Figure 5.11: Test runs of trained peephole LSTM networks on the f_{rect} PFG task with $F=10$ (left) and $F=25$ (right). Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

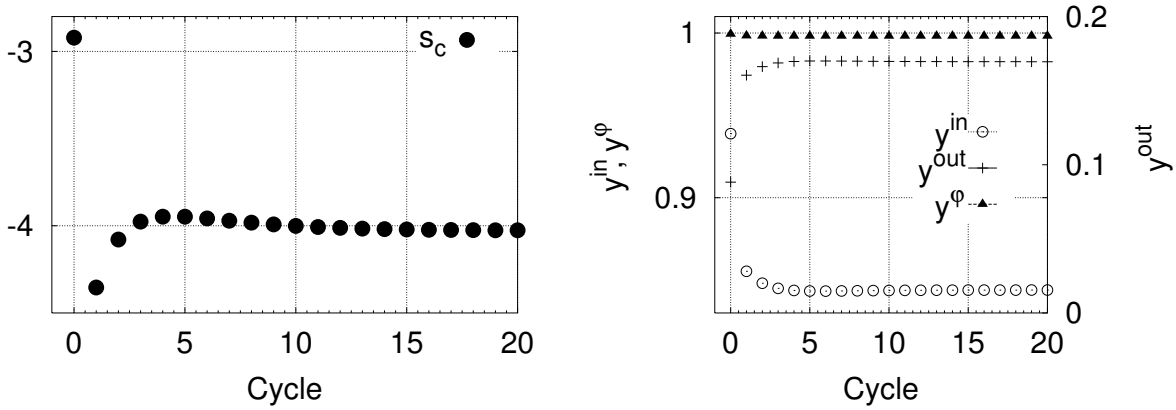

Figure 5.12: Cell states and gate activations at the onset (zero phase) of the first 20 cycles during a test run with a trained LSTM network on the $f_{\cos}$ PFG task with $F = 10$. Note that the initial state (at cycle 0) is quite far from the equilibrium state.

5.5.5 General Observation: Network initialization

At the beginning of each stream cell states and gate activations are initialized to zero. This initial state is almost always quite far from the corresponding state in the same phase of later periods in the stream. Figure 5.12 illustrates this for the $f_{\cos}$ task. After few consecutive periods, cell states and gate activations of successful networks tend to settle to very stable, phase-specific values, which are typically quite different from the corresponding values in the first period. This suggests that the initial state of the network should be learned as well, as proposed by Forcada and Carrasco (1995), instead of arbitrarily initializing it to zero.

5.6 Conclusion

Previous work on LSTM did not require the network to extract relevant information conveyed by the duration of intervals between events. Here we show that LSTM can solve such highly nonlinear tasks as well, by learning to precisely measure time intervals, provided we furnish LSTM cells with peephole connections that allow them to inspect their current internal states. It is remarkable that peephole LSTM can learn exact and extremely robust timing algorithms without teacher forcing, even in case of very uninformative, rarely changing target signals. This makes it a promising approach for numerous real-world tasks whose solution partly depend on the precise duration of intervals between relevant events.

Chapter 6

Simple Context Free and Context Sensitive Languages

6.1 Introduction

Previous work showed that LSTM outperforms traditional RNN algorithms on tasks that require to learn the rules of regular languages (RLs), see Chapter 3 and Hochreiter and Schmidhuber (1997). RLs are describable by deterministic finite state automata (DFA) (Casey, 1996; Siegelmann, 1992; Blair & Pollack, 1997; Kalinke & Lehmann, 1998; Zeng, Goodman, & Smyth, 1994"). Until now, however, it has remained unclear whether LSTM's superiority carries over to tasks involving context free languages (CFLs), such as those discussed in the RNN literature (Sun, Giles, Chen, & Lee, 1993; Wiles & Elman, 1995; Steijvers & Grunwald, 1996; Tonkes & Wiles, 1997; Rodriguez, Wiles, & Elman, 1999; Rodriguez & Wiles, 1998). Their recognition requires the functional equivalent of a stack. It is conceivable that LSTM has just the right bias for RLs but might fail on CFLs.

Here we will focus on the most common CFLs benchmarks found in the RNN literature: $a^n b^n$ and $a^n b^m B^m A^n$. We study questions such as:

- Can LSTM learn the functional equivalent of a pushdown automaton?
- Given training sequences up to size n, can it generalize to $n+1, n+2, \ldots$?
- How stable are the solutions?
- Does LSTM outperform previous approaches?

Finally we will apply LSTM to a context *sensitive* language (CSL). The CSLs include the CFLs, which include the RLs. We will focus on the classic example $a^n b^n c^n$, which is a CSL but not a CFL (Section 6.2). In general, CSL recognition requires a linear-bounded automaton, a special Turing machine whose tape length is at most linear in the input size. The $\{a^n b^n c^n\}$ language is one of the simplest CSLs; it can be generated by a tree-adjoined grammar and recognized using a so-called embedded push-down automaton (Vijay-Shanker, 1992) or a finite

state automaton with access to two counters that can be incremented or decremented. To our knowledge no RNN has been able to learn a CSL.

We are using LSTM with forget gates and peephole connections introduced in the previous chapters.

6.2 Experiments

The network sequentially observes exemplary symbol strings of a 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 have to be predicted, and none of the others.
The network sequentially observes exemplary symbol strings of a given language, presented one input symbol at a time, also referred to as input sequences. Every input sequence begins with the start symbol S. The empty string, consisting of ST only, is considered part of each language. A string is accepted when all predictions have been correct. Otherwise it is rejected.
This prediction task is equivalent to a classification task with two classes "accept" and "reject," because the system will make prediction errors for all strings outside the language. A system has learned a given language up to string size n once it is able to correctly predict all strings with size $\leq n$.

Symbols are encoded locally by d-dimensional binary vectors with only one non-zero component, where d equals the number of language symbols plus one for either the start symbol in the input or the "end of string" symbol in the output (d input units, d output units). $+1$ signifies that a symbol is set and -1 that it is not; the decision boundary for the network output is 0.0.

CFL $a^n b^n$ (Sun et al., 1993; Wiles & Elman, 1995; Tonkes & Wiles, 1997; Rodriguez et al., 1999). Here the strings in the input sequences are of the form $a^n b^n$; input and output vectors are 3-dimensional. Prior to the first occurrence of b either a or b, or a or T at sequence beginnings, are possible in the next step. Thus, e.g., for $n=5$:

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

An example for a set of context-free production rules for the $a^n b^n$ grammar is: S $\rightarrow \alpha \mid \epsilon$; $\alpha \rightarrow$ aαb $\mid \epsilon$, where S is the starting symbol, α is a non-terminal symbol and ϵ is the empty string.

CFL $a^n b^m B^m A^n$ (Rodriguez & Wiles, 1998). The second half of a string from this palindrome or mirror language is completely predictable from the first half. The task involves an intermediate time lag of length $2m$. Input and output vectors are 5-dimensional. Prior to the first occurrence of B two symbols are possible in the next step. Thus, e.g., for $n=4, m=3$:

Input:	S	a	a	a	a	b	b	b	B	B	B	A	A	A	A
Target:	a/T	a/b	a/b	a/b	a/b	b/B	b/B	b/B	B	B	A	A	A	A	T

The $a^n b^m B^m A^n$ grammar can be produced by similar context-free rules as the $a^n b^n$ grammar using two non-terminal symbols (α and β): S $\rightarrow \alpha \mid \epsilon$; $\alpha \rightarrow$ aαA $\mid \epsilon \mid \beta$; $\beta \rightarrow$ bβB $\mid \epsilon$.

CSL $a^n b^n c^n$. Input and output vectors are 4-dimensional. Prior to the first occurrence of b two symbols are possible in the next step. Thus, e.g., for $n=5$:

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

The pumping Lemma for context-free languages can be applied to show that $a^n b^n c^n$ is not context-free. An intuitive explanation is that it is necessary to consider the number of a symbols then producing b and c symbols, this requires context information.

6.2.1 Training and Testing

Learning and testing alternate: after each epoch (= 1000 training sequences) we freeze the weights and run a test. Even when all strings are processed correctly during training, it is necessary to test again with frozen weights once all weight changes have been executed. Apart from ensuring the learning of the training set the test also determines generalization performance, which we did not optimize by using, say, a validation set.

Training and test sets incorporate all legal strings up to a given length: $2n$ for $a^n b^n$, $3n$ for $a^n b^n c^n$ and $2(n+m)$ for $a^n b^m B^m A^n$. Training strings are presented in random order. Only exemplars from the class "accept" are presented. Training is stopped once all training sequences have been accepted, or after at most 10^7 training sequences. The *generalization set* is the largest accepted test set (assuming that the network generalizes at all).
Weight changes are made after each sequence. We apply the momentum algorithm (Plaut et al., 1986) with learning rate α is 10^{-5} and momentum parameter 0.99. All results are averages over 10 independently trained networks with different weight initializations (these 10 initializations are identical for each experiment).

CFL $a^n b^n$. We study training sets with $n \in \{1,..,N\}$. We test all sets with $n \in \{1,..,M\}$ and $M \in \{N,..,1000\}$ (sequences of length ≤ 2000).

CFL $a^n b^m B^m A^n$. We use two training sets: a) The same set as used by Rodriguez and Wiles (1999) : $n \in \{1,..,11\}$, $m \in \{1,..,11\}$ with $n+m \leq 12$ (sequences of length ≤ 24). b) The set given by $n \in \{1,..,11\}$, $m \in \{1,..,11\}$ (sequences of length ≤ 44). We test all sets with $n \in \{1,..,M\}$, $m \in \{1,..,M\}$ and $M \in \{11,..,50\}$ (sequences of length ≤ 200).

CSL $a^n b^n c^n$. We study two kinds of training sets: a) with $n \in \{1,..,N\}$ and b) with $n \in \{N-1, N\}$. Case b) asks for a major generalization step that seems almost impossible at first glance: Given very similar training sequences whose sizes differ by at most 2, learn to process sequences of arbitrary size! We test all sets with $n \in \{L,..,M\}$, $L \in \{1,..,N-1\}$ and $M \in \{N,..,500\}$ (sequences of length ≤ 1500).

6.2.2 Network Topology and Experimental Parameters

The input units are fully connected to a hidden layer consisting of memory blocks with 1 cell each. The cell outputs are fully connected to the cell inputs, to all gates, and to the output units, which also have direct "shortcut" connections from the input units (Figure 6.1). For each task we selected the topology with minimal number of memory blocks that solved the task without extensive parameter optimization. Larger topologies never led to disadvantages except for an increase in computational complexity.

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. Although not critical, these values have been found empirically to work well; we use them for all our experiments. The forget gates start off closed, so that the cells initially remember everything. We also tried different bias configurations; the results were qualitatively the same, which supports our claim that precise initialization is not critical. All other weights are initialized randomly in the range $[-0.1, 0.1]$. The cell's input squashing function g is the identity function. The squashing function of the output units is a sigmoid function with the range $[-2, 2]$.

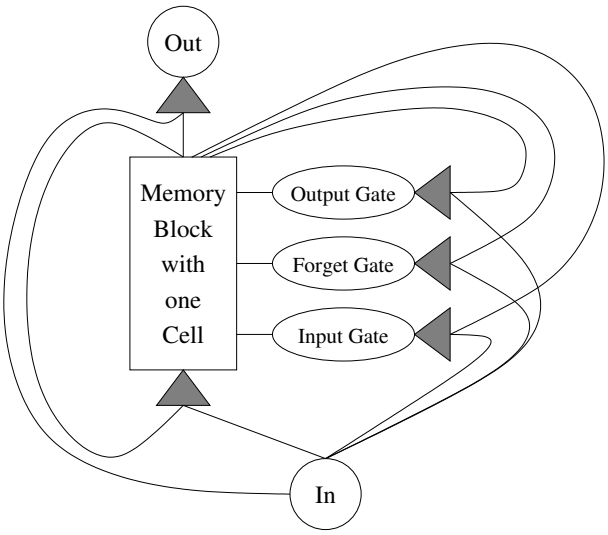

Figure 6.1: Three-layer LSTM topology with a single input and output. Recurrence is limited to the hidden layer, consisting here of a single LSTM memory block with a single cell. All 10 "unit-to-unit" connections are shown (but bias and peephole connections are not).

Reference	Hidden Units	Train. Set $[n]$	Train. Str. $[10^3]$	Sol./ Tri.	Best Test $[n]$
(Sun et al., 1993)[1]	5	1,..,160	13.5	1/1	1,..,160
(Wiles & Elman, 1995)	2	1,..,11	2000	4/20	1,..,18
(Tonkes & Wiles, 1997)	2	1,..,10	10	13/100	1,..,12
(Rodriguez et al., 1999)[2]	2	1,..,11	267	8/50	1,..,16

Table 6.1: Previous results for the CFL $a^n b^n$, showing (from left to right) the number of hidden units or state units, the values of n used during training, the number of training sequences, the number of found solutions/trials and the largest accepted test set.

CFL $a^n b^n$. We use one memory block (with one cell). With peephole connections there are 38 adjustable weights (3 peephole, 28 unit-to-unit and 7 bias connections).

CFL $a^n b^m B^m A^n$. We use two blocks with one cell each, resulting in 110 adjustable weights (6 peephole, 91 unit-to-unit and 13 bias connections).

CSL $a^n b^n c^n$. We use the same topology as for the $a^n b^m B^m A^n$ language, but with 4 input and output units instead of 5, resulting in 90 adjustable weights (6 peephole, 72 unit-to-unit and 12 bias connections).

6.2.3 Previous results

CFL $a^n b^n$. Published results on the $a^n b^n$ language are summarized in Table 6.1. RNNs trained

[1] Sun's training set was augmented stepwise by sequences misclassified during testing, and in the final accepted set n was in {1,.., 20} except for 20 random sequences up to length $n=160$ (the exact generalization performance was unclear).

[2] Applying brute force search to the weights of the best network of Rodriguez et al. (1999) further improves performance to acceptance up to $n=28$.

Train. Set [n]	Train. Str. [10^3]	% Sol.	Generalization Set [n]
1,..,10	22 (19)	100	1,..,1000 (1,..,118)
1,..,20	18 (19)	100	1,..,587 (1,..,148)
1,..,30	16 (19)	100	1,..,1000 (1,..,408)
1,..,40	25 (28)	100	1,..,1000 (1,..,628)
1,..,50	42 (40)	100	1,..,767 (1,..,430)

Table 6.2: Results for the $a^n b^n$ language, showing (from left to right) the values for n used during training, the average number of training sequences until best generalization was achieved, the percentage of correct solutions and the best generalization (average over all networks given in parenthesis).

with plain BPTT tend to learn to just reproduce the input (Wiles & Elman, 1995; Tonkes & Wiles, 1997; Rodriguez et al., 1999). Sun et al. (1993) used a highly specialized architecture, the "neural pushdown automaton", which also did not generalize well (Sun et al., 1993; Das, Giles, & Sun, 1992).

CFL $a^n b^m B^m A^n$. Rodriguez and Wiles (1998) used BPTT-RNNs with 5 hidden nodes. After training with $51 \cdot 10^3$ strings with $n + m \leq 12$ (sequences of length ≤ 24), most networks generalized on longer off-training set strings. The best network generalized to sequences up to length 36 ($n = 9$,$m = 9$). But none of them learned the complete training set.

CSL $a^n b^n c^n$. To our knowledge no previous RNN ever learned a CSL.

6.2.4 LSTM Results

CFL $a^n b^n$. 100% solved for all training sets (Table 6.2). Small training sets ($n \in \{1,..,10\}$) were already sufficient for perfect generalization up to the tested maximum: $n \in \{1,..,1000\}$. Note that long sequences of this kind require very stable, finely tuned control of the network's internal counters (Casey, 1996).
This performance is much better than that of previous approaches, where the largest set was learned by the specially designed neural push-down automaton (Sun et al., 1993; Das et al., 1992): $n \in \{1,..,160\}$. The latter, however, required training sequences of the same length as the test sequences. From the training set with $n \in \{1,..,10\}$ LSTM generalized to $n \in \{1,..,1000\}$, whereas the best previous result (see Table 6.1) generalized only to $n \in \{1,..,18\}$ (even with a slightly larger training set: $n \in \{1,..,11\}$). In contrast to Tonkes and Wiles (1997), we did not observe our networks forgetting solutions as training progresses. So unlike all previous approaches, LSTM reliably finds solutions that generalize well.

The fluctuations in generalization performance for different training sets in Table 6.2 may be due to the fact that we did not optimize generalization performance by using a validation set. Instead we simply stopped each epoch (= 1000 sequences) once the training set was learned.

CFL $a^n b^m B^m A^n$. Training set a): 100% solved; after $29 \cdot 10^3$ training sequences the best network of 10 generalized to at least $n, m \in \{1,..,22\}$ (all strings up to a length of 88 symbols processed correctly); the average generalization set was the one with $n, m \in \{1,..,16\}$ (all strings up to a length of 64 symbols processed correctly), learned after $25 \cdot 10^3$ training sequences on average.

Training set b): 100% solved; after $26 \cdot 10^3$ training sequences the best network generalized

Train. Set [n]	Train. Str. [10^3]	% Sol.	Generalization Set [n]
1,..,10	54 (62)	100	1,..,52 (1,..,28)
1,..,20	28 (43)	100	1,..,160 (1,..,66)
1,..,30	37 (43)	100	1,..,228 (1,..,91)
1,..,40	51 (48)	90	1,..,500 (1,..,120)
1,..,50	60 (94)	100	1,..,500 (1,..,409)
10,11	24 (78)	100	9,..,12 (10,..,11)
20,21	829 (626)	40	10,..,27 (17,..,23)
30,31	42 (855)	30	29,..,34 (29,..,32)
40,41	854 (1597)	40	20,..,57 (35,..,45)
50,51	32 (621)	60	43,..,57 (47,..,55)

Table 6.3: Results for the $a^n b^n c^n$ language, showing (from left to right) the values for n used during training, the average number of training sequences until best generalization was achieved, the percentage of correct solutions and the best generalization (average over all networks in parenthesis).

to at least $n, m \in \{1,..,23\}$ (all strings until a length of 92 symbols processed correctly). The average generalization set was the one with $n, m \in \{1,..,17\}$ (all strings until a length of 68 symbols processed correctly), learned after $82 \cdot 10^3$ training sequences on average. Unlike the previous approach of Rodriguez and Wiles (1998), LSTM easily learns the complete training set and reliably finds solutions that generalize well.

CSL $a^n b^n c^n$. LSTM learns 4 of the 5 training sets in 10 out of 10 trials (only 9 out of 10 for the training set with $n \in \{1,..,40\}$) and generalizes well (Table 6.3). Small training sets ($n \in \{1,..,40\}$) were already sufficient for perfect generalization up to the tested maximum: $n \in \{1,..,500\}$, that is, sequences of length up to 1500. Even in absence of any short training sequences ($n \in \{N-1, N\}$) LSTM learned well (see bottom half of Table 6.3).

We also modified the training procedure, by presenting each exemplary string without providing all possible next symbols as targets, but only the symbol that actually occurs in the current exemplar. This led to slightly longer training durations, but did not significantly change the results.

6.2.5 Analysis

How do the solutions discovered by LSTM work?

CFL $a^n b^n$. Figure 6.2 shows a test run with a network solution for $n = 5$. The cell state s_c increases while a symbols are fed into the network, then decreases (with the same step size) while b symbols are fed in. At sequence beginnings (when the first a symbols are observed), however, the step size is smaller due to the closed input gate, which is triggered by s_c itself. This results in "overshooting" the initial value of s_c at the end of a sequence, which in turn triggers the opening of the output gate, which in turn leads to the prediction of the sequence termination.

CFL $a^n b^m B^m A^n$. The behavior of a typical network solution is shown in Figure 6.3. The network learned to establish and control two counters. The two symbol pairs (a, A) and (b, B) are treated separately by two different cells, c_2 and c_1, respectively. Cell c_2 tracks the difference

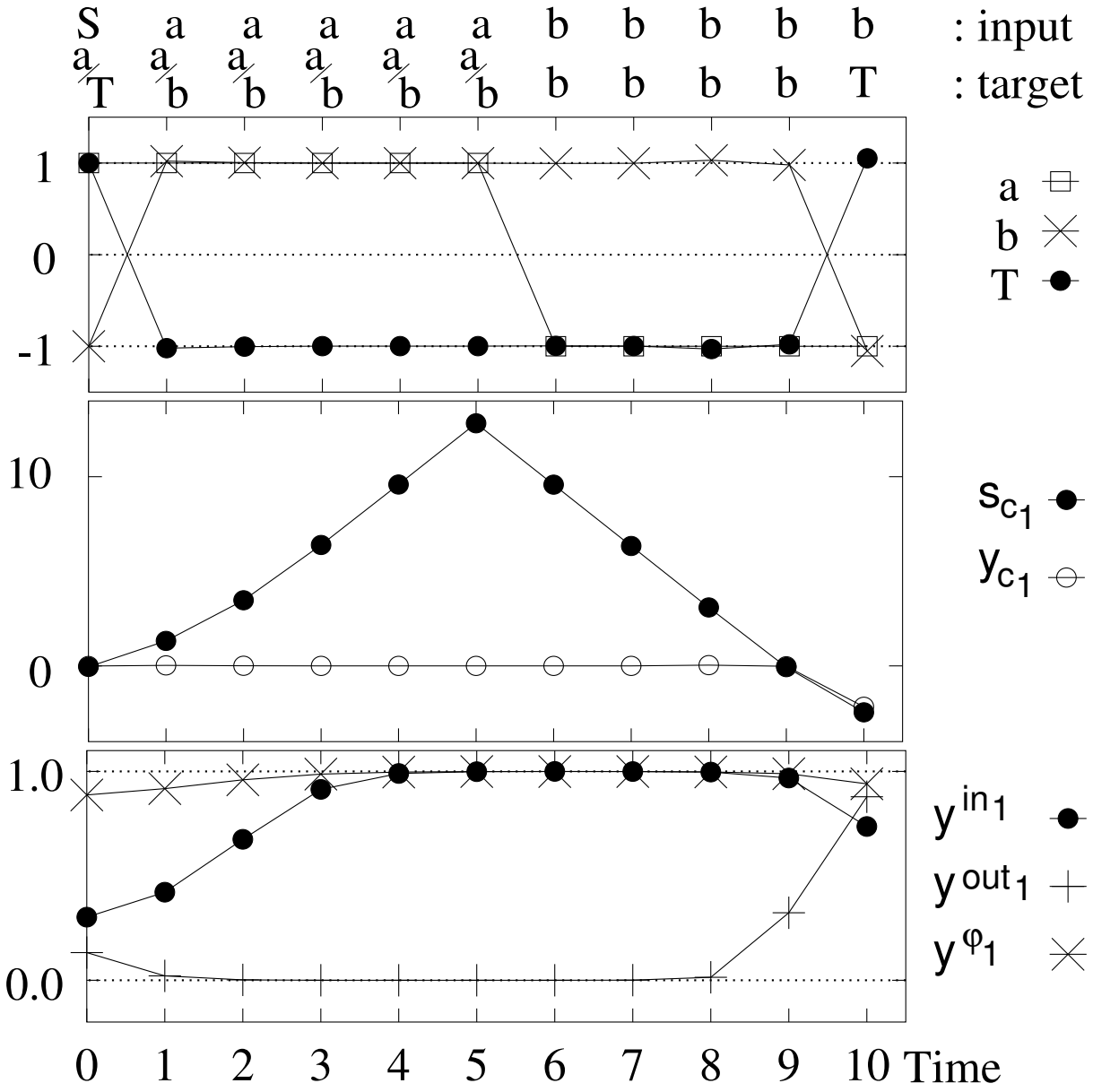

Figure 6.2: CFL $a^n b^n$ ($n = 5$): Test run with network solutions. Top: Network output y_k. Middle: Cell state s_c and cell output y_c. Bottom: Activations of the gates (input gate y_{in}, forget gate y_φ and output gate y_{out}).

between the number of observed a and A symbols. It opens only at the end of a string, where it predicts the final T. Cell c_1 treats the embedded $b^m B^m$ substring in a similar way. While values are stored and manipulated within a cell, the output gate remains closed. This prevents the cell from disturbing the rest of the network and also protects its CEC against incoming errors.

CSL $a^n b^n c^n$. The network solutions use a combination of two counters, instantiated separately in the two memory blocks (Figure 6.4). Here the second cell counts up, given an a input symbol. It counts down, given a b. A c in the input causes the input gate to close and the forget gate to reset the cell state s_c. The second memory block does the same for b, c, and a,

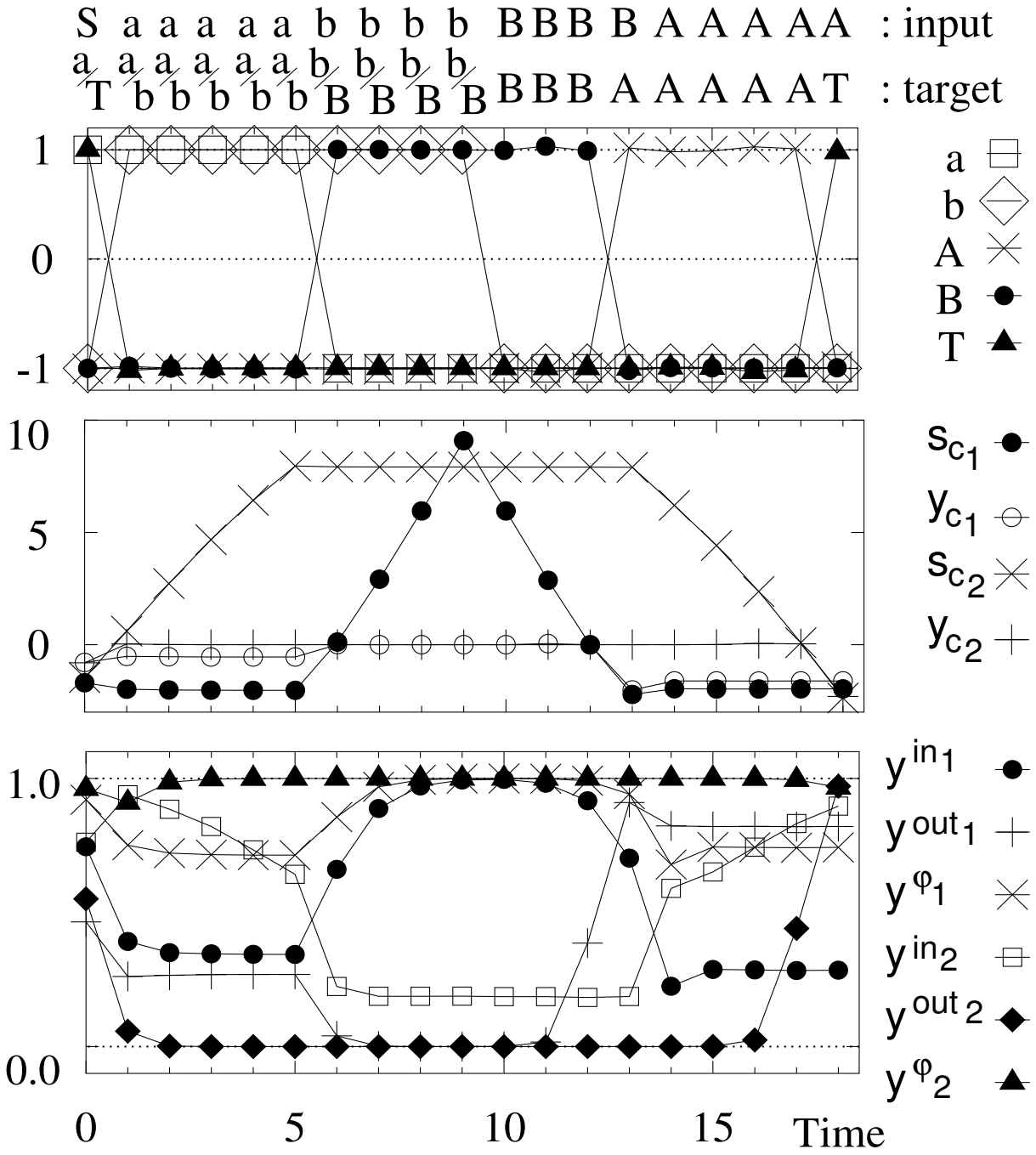

Figure 6.3: CFL $a^n b^m B^m A^n$ $(n = 5, m = 4)$: Test run with network solution. Top: Network output y_k. Middle: Cell state s_c and cell output y_c. Bottom: Activations of the gates (input gate y_{in}, forget gate y_φ and output gate y_{out}).

respectively. The opening of output gate of the first block indicates the end of a string (and the prediction of the last T), triggered via its peephole connection.

Why does the network not generalize for short strings when using only two training strings as for the $a^n b^n c^n$ language (see Table 6.3)? The gate activations in Figure 6.4 show that activations

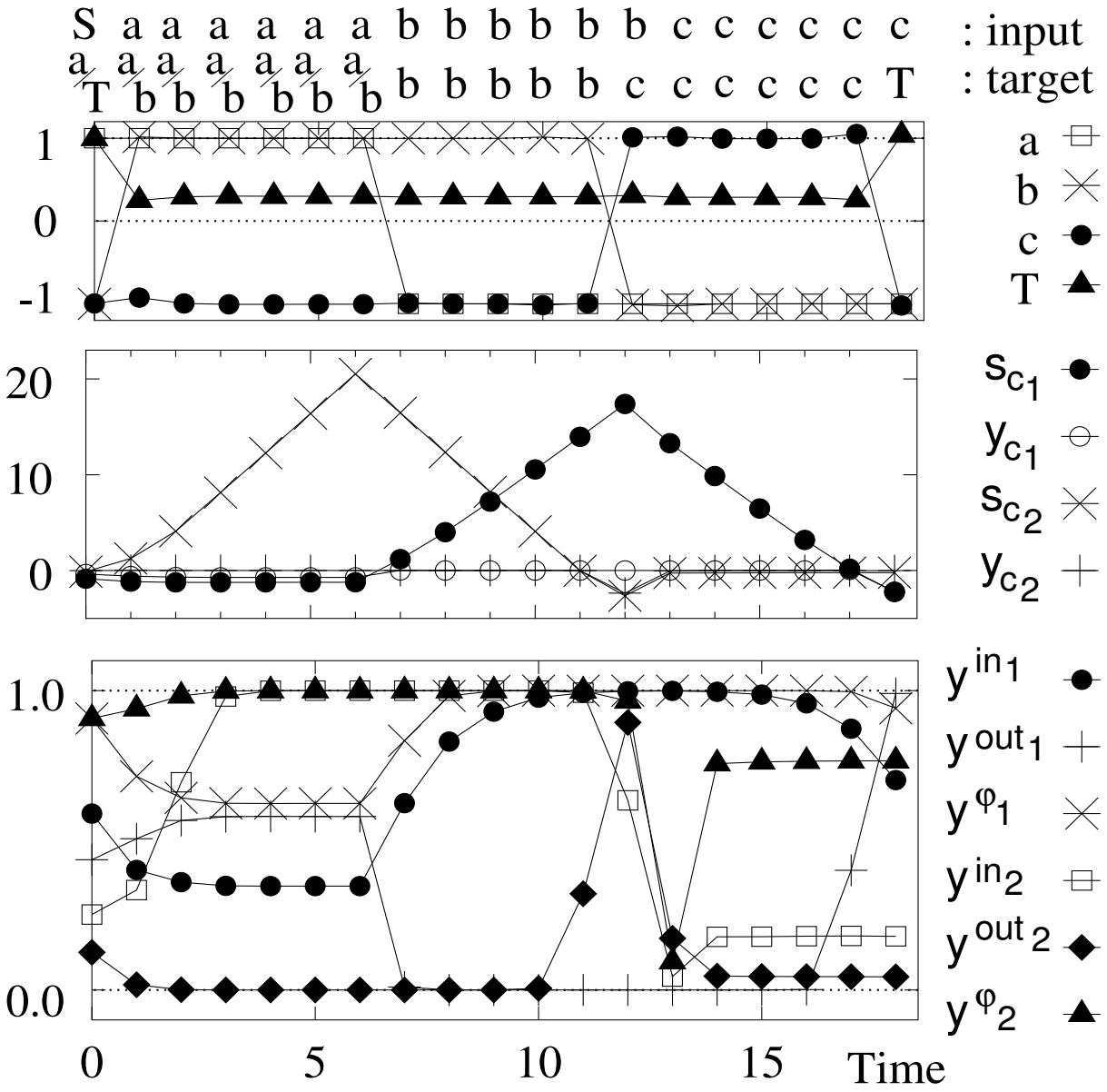

Figure 6.4: CSL $a^n b^n c^n$ ($n = 5$): Test run with network solution (the system scales up to sequences of length 1000 and more). Top: Network output y_k. Middle: Cell state s_c and cell output y_c. Bottom: Activations of the gates (input gate y_{in}, forget gate y_φ and output gate y_{out}).

slightly drift even when the input stays constant. Solutions take this state drift into account, and will not work without it or with too much of it, as in the case when the sequences are much shorter or longer than the few observed training examples. This imposes a limit on generalization in *both* directions (towards longer and shorter strings). We found solutions with less drift to generalize better.

Further improvements. Even better results can be obtained through increased training time and stepwise reduction of the learning rate, as done in (Rodriguez et al., 1999). The distribution of lengths of sequences in the training set also affects learning speed and generalization.

A set containing more long sequences improves generalization for longer sequences. Omitting the sequence with $n=1$ (and $m=1$), typically the last one to be learned, has the same effect. Training sets with many short and many long sequences are learned more quickly than uniformly distributed ones.

Related tasks. The $(ba^k)^n$ regular language is related to $a^n b^n$ in the sense that it requires to learn a counter, but the counter never needs counting down. This task is equivalent to the "Generating timed spikes" task (Section 5.5.3) learned by LSTM for $k = 50$ with $n \geq 1000$. A hand-made, hardwired solution (no learning) of a second order RNN worked for values of k until 120 (Steijvers & Grunwald, 1996).

For all three tasks peephole connections are mandatory. The output gates remain closed for substantial time periods during each input sequence presentation (compare Figures 6.2, 6.3 and 6.4); the end of such a period is always triggered via peephole connections.

6.3 Conclusion

We found that LSTM clearly outperforms previous RNNs not only on regular language benchmarks (according to previous research) but also on context free language (CFL) benchmarks; it learns faster and generalizes better. LSTM also is the first RNN to learn a context sensitive language.

Although CFLs like those studied here may also be learnable by certain discrete symbolic grammar learning algorithms (SGLAs) (Sakakibara, 1997; Lee, 1996; Osborne & Briscoe, 1997), the latter exhibit more task-specific bias, and are not designed to solve numerous other sequence processing tasks involving noise, real-valued inputs / internal states, and continuous output trajectories, which LSTM solves easily (see previous chapters and Hochreiter and Schmidhuber (1997)). SGLAs include a large range of methods, such as decision-tree algorithms (see e.g., Quinlan (1992)), case-based and explanation-bases reasoning (see e.g., Mitchell, Keller, and Kedar-Cabelli (1986), Porter, Bruce, Bareiss, and Holte (1990)), and inductive logic programming (see e.g., Zelle and Mooney (1993)).

Our findings reinforce the perception that LSTM is a very general and promising adaptive sequence processing device, with a wider field of potential applications than alternative RNNs.

Chapter 7

Time Series Predictable Through Time-Window Approaches

7.1 Introduction

In the previous chapters we have applied LSTM to numerous temporal processing tasks, such as: continual grammar problems, recognition of temporally extended, noisy patterns (Chapter 3); arithmetic operations on continual input streams and robust storage of real numbers across extended time intervals (Chapter 4); extraction of information conveyed by the temporal distance between events and generation of precisely timed events (Chapter 5); stable generation of smooth and highly nonlinear periodic trajectories (Chapter 5); Recognition of regular and context free and context sensitive languages (Chapter 6).

Time series benchmark problems found in the literature, however, often are conceptually simpler than the above. They often do not require RNNs at all, because all relevant information about the next event is conveyed by a few recent events contained within a small time window. Here we apply LSTM to such relatively simple tasks, to establish a limit to the capabilities of the LSTM-algorithm in its current form. We focus on two intensively studied tasks, namely, prediction of the Mackey-Glass series (Mackey & Glass, 1977) and chaotic laser data (Set A) from a contest at the Santa Fe Institute (1992).

LSTM is run as a "pure" autoregressive (AR) model that can only access input from the current time-step, reading one input at a time, while its competitors — e.g., multi-layer perceptrons (MLPs) trained by back-propagation (BP) — simultaneously see several successive inputs in a suitably chosen time window. Note that Time-Delay Neural Networks (TDNNs) (Haffner & Waibel, 1992) are not purely AR, because they allow for direct access to past events. Neither are NARX networks (Lin et al., 1996) which allow for several distinct input time windows (possibly of size one) with different temporal offsets.

We also evaluate stepwise versus iterated training as proposed by Principe and Kuo (1995) to make RNNs learn a dynamic attractor rather than simply approximate output. It was found by Principe, Rathie, and Kuo (1992) that neural networks trained with iterative training outperform traditional prediction algorithms in approximating "real" chaotic attractors. Bakker, Schouten,

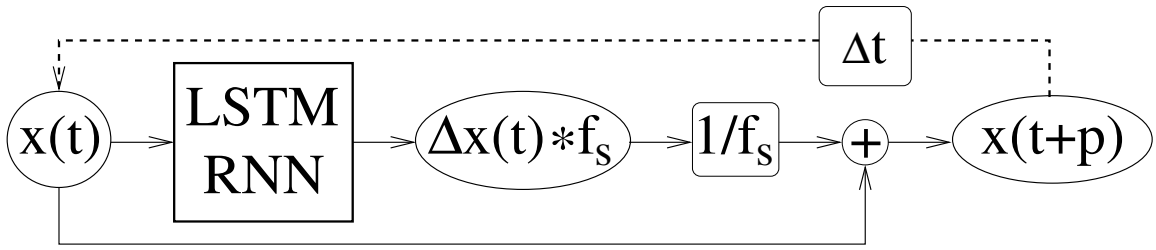

Figure 7.1: AR-RNN setup for time series prediction.

Giles, Takens, and Bleek (2000) refined the iterated training scheme and found it superior to stepwise training. Here we cannot generally confirm this result.

7.2 Experimental Setup

The task is to use currently available points in the time series to predict the future point, $t+T$. The target for the network t_k is the difference between the values $x(t+p)$ of the time series p steps ahead and the current value $x(t)$ multiplied by a scaling factor f_s: $t_k(t) = f_s \cdot (x(t+p) - x(t)) = f_s \cdot \Delta x(t)$. f_s scales $\Delta x(t)$ between –1 and 1 for the training set, the same value for f_s is used during testing. The predicted value is the network output divided by f_s plus $x(t)$ (Figure 7.1). During iterated prediction with $T = n * p$ the output is clamped to the input (self-iteration) and the predicted values are fed back n times. For direct prediction $p = T$ and $n = 1$; for single-step prediction $p = 1$ and $n = T$.

Note that during iterated prediction, the network state after the first prediction has to be stored and re-established after the last self-iterations. For the iterated prediction with $p > 1$ and $n > 1$ the setup becomes more complex: p copies of the network have to predict in parallel. The network predicting $x(t + n \cdot p)$ starts with $x(t)$, and feeds back the predicted values $n-1$ times to the input before the same procedure is executed with a second network starting with $x(t+1)$ at the task of predicting $x(t + 1 + n \cdot p)$. The internal network state is indirectly also trained to move from $s(t)$ to $s(t+p)$ in one iteration. Hence, the iterated prediction of one series with step-size $p > 1$ results in the parallel prediction of p series with p different starting points: $t_{start} = 0, 1, 2, \ldots, p-1$. For example, given $T = 84$ and $p = 6$ ($\Rightarrow n = 14$), we start at $t_{start} = 0$ and iterate 14 times to predict the value at $t = 84$. We then use another copy of the network to predict $t = 84 + 1$ starting at $t_{start} = 1$ and so forth.

Bakker, Schouten, Giles, Takens, and Bleek (2000) proposed to mix network predictions with the target values during iterated training. One challenge with this procedure lies in finding the right mixing coefficient. Bakker et al. used the same constant value throughout training. This procedure has the disadvantage that bad predictions at the beginning of the training induce a lot of "input noise". We modified Bakker's idea by introducing a maximum output error e_{max} for iterated training in place of a mixture. When the error at the output e_k was larger than $e_{max} = 0.5$, the output was unclamped and training continued with the next true input value at $t + 1 + p$.

This scheme has the advantage that the number of iterated steps is coupled to training performance and is in this way self-regularizing. In preliminary experiments we tested our method using constant prediction-target mixtures having different coefficients. Our method always learned faster and with fewer network divergences.

The error measure is the normalized root mean squared error: NRMSE $= \langle (y_k - t_k)^2 \rangle^{\frac{1}{2}} / \langle (t_k -$

$\langle t_k \rangle)^2 \rangle^{\frac{1}{2}}$, where y_k is the network output and t_k the target. The reported performance is the best result of 10 independent trials.

7.2.1 Network Topology

LSTM. The input units are fully connected to a hidden layer consisting of memory blocks with 1 cell each. The cell outputs are fully connected to the cell inputs, to all gates, and to the output units. All gates, the cell itself and the output unit are biased. Bias weights to input and output gates are initialized block-wise: -0.5 for the first block, -1.0 for the second, -1.5 for the third, and so forth. Forget gates are initialized with symmetric positive values: $+0.5$ for the first block, $+1$ for the second block, etc. We also tried a bias configuration with reversed signs for the initial values. In this case the gates are open (so no gating) and the cells forget almost immediately. This configuration is similar to a RNN with one fully recurrent hidden layer. The results were qualitatively the same, which supports our claim that precise initialization is not critical. All other weights are initialized randomly in the range $[-0.1, 0.1]$. The cell's input squashing function g is a sigmoid function with the range $[-1, 1]$. The squashing function of the output units is the identity function.

To have statistically independent weight updates, we execute weight changes every $50 +$ rand(50) steps (where rand(max) stands for a random positive integer smaller than max which changes after every update). We use a constant learning rate $\alpha = 10^{-4}$.

MLP. The MLPs we use for comparison have one hidden layer and are trained with BP. As with LSTM, the one output unit is linear and Δx is the target. The input differs for each task but in general uses a time window with a time-space embedding. All units are biased and the learning rate is $\alpha = 10^{-3}$.

Note that we do not use IO shortcuts, because they become short circuits during self iteration, causing exponential growth of the output unit's activity.

7.3 Mackey-Glass Chaotic Time Series

The Mackey-Glass chaotic time series (Mackey & Glass, 1977) can be generated from the Mackey-Glass delay-differential equation:

$$\dot{x}(t) = \frac{\alpha x(t-\tau)}{1 + x^c(t-\tau)} - \beta x(t) \tag{7.1}$$

We generate benchmark sets using the parameters $a = 0.2$, $b = 0.1$, $c = 10$ and $\tau = 17$. For $\tau > 16.8$ the series becomes chaotic. $\tau = 17$ results in a quasi-periodic series with a characteristic period $T_c \approx 50$, lying on an attractor with fractal dimension $D = 2.1$. To generate these benchmark sets, 7.1 is integrated using a four-point Runge-Kutta method with step size 0.1 and the initial condition $x(t) = 0.8$ for $t < 0$. The equation is integrated up to $t = 5500$, with the points from $t = 200$ to $t = 3200$ used for training and the points from $t = 5000$ to $t = 5500$ used for testing.

Figure 7.2 shows the first 100 points from the test set. Since the Mackey-Glass time series is chaotic, it is difficult to predict for values of T greater than its characteristic period T_c of approximately 50. In the literature a number of different prediction points have been tried: $T \in \{1, 6, 84, 85, 90\}$. For the comparison of results we consider the predictions with offsets $T \in \{84, 85, 90\}$ as equal tasks. For approaches that use as input a time window of past values it is common to use the four delays t, $t-6$, $t-12$ and $t-18$. These points represent an adequate delay-state embedding for the prediction of Mackey-Glass series assuming $T = 6$. For further

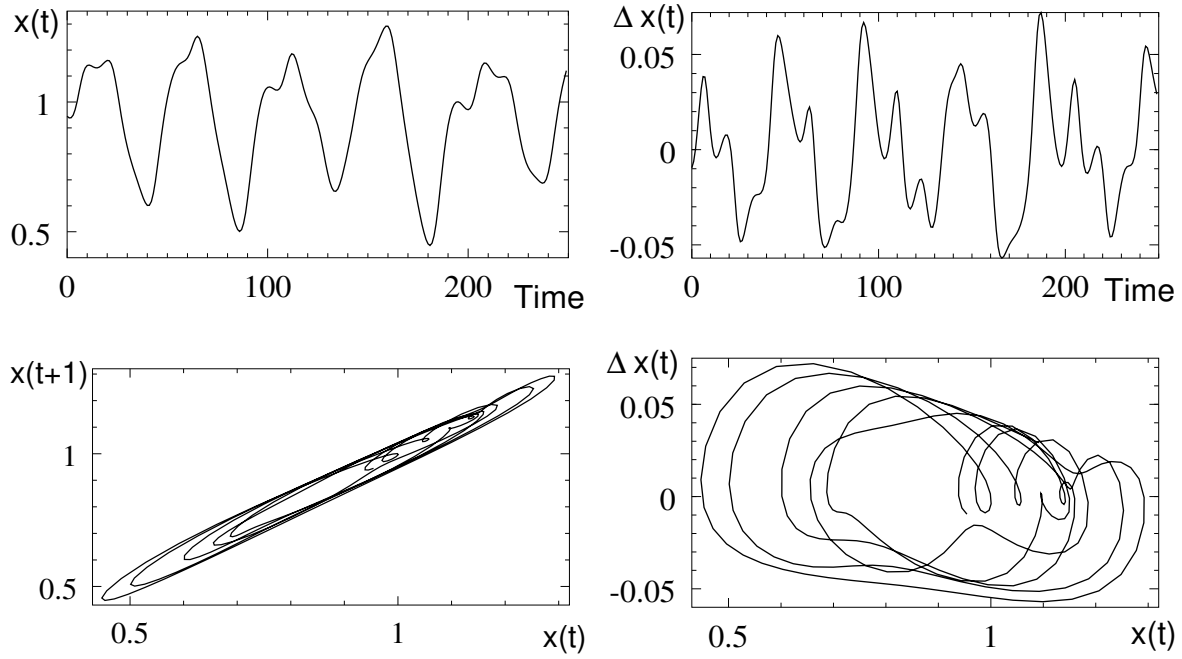

Figure 7.2: Mackey-Glass time series (test set). Top-Left: Cut-out of the series. Top-Right: The first difference for $p = 1$: $\Delta x(t) = (x(t+1) - x(t))$. Bottom-Left: $x(t+1)$ against $x(t)$. Bottom-Right: $\Delta x(t)$ against $x(t)$.

explanation see, for example, Falco, Iazzetta, Natale, and Tarantino (1998"). As explained above, LSTM received only the value of $x(t)$ as input.

7.3.1 Previous Work

In the following sections we attempt to summarize existing attempts to predict these time series. To allow comparison among approaches, we did not consider works where noise was added to the task or where training conditions were very different from ours. When not specifically mentioned, an input time window with time delays t, $t-6$, $t-12$ and $t-18$ or larger was used. The different approaches are outlined in Table 7.1. Vesanto (1997) offers the best result to date, according to our knowledge, with a Self-Organizing Map (SOM) approach. The SOM parameters given in Table 7.2 refers to the prototype vectors of the map. The results from these approaches are found in Table 7.2. We re-calculated the results for R. Bone et al., because only the NMSE was given.

7.3.2 Results

The LSTM results are listed at the bottom of Table 7.2. After six single-steps of iterated training ($p=1$, $T=6$, $n=6$) the LSTM NRMSE for single step prediction ($p=T=1$, $n=1$) is: 0.0452. After 84 single-steps of iterated training ($p=1$, $T=84$, $n=84$) the LSTM NRMSE for single step prediction ($p=T=1$, $n=1$) is: 0.0809. Figure 7.3 shows iterated prediction results for LSTM. Increasing the number of memory blocks did not significantly improve the results.

Why did LSTM perform worse than the MLP? The AR-LSTM network does not have access to the past as part of its input and therefore has to learn to extract and represent a Markov

Model	Author	Description
BPNN	Day and Davenport (1993)	A BP continuous-time feed forward NNs with two hidden layers and with fixed time delays.
ATNN	Day and Davenport (1993)	A BP continuous-time feed forward NNs with two hidden layers and with adaptable time delays.
DCS-LMM	Chudy and Farkas (1998)	Dynamic Cell Structures combined with Local Linear Models.
EBPTTRNN	R. Bone, Crucianu, Verley, and Asselin de Beauville (2000)	RNNs with 10 adaptive delayed connections trained with BPTT combined with a constructive algorithm.
BGALR	Falco, Iazzetta, Natale, and Tarantino (1998")	A genetic algorithm with adaptable input time window size (Breeder Genetic Algorithm with Line Recombination).
EPNet	Yao and Liu (1997)	Evolved neural nets (Evolvable Programming Net).
SOM	Vesanto (1997)	A Self-organizing map.
Neural Gas	Martinez, Berkovich, and Schulten (1993)	The Neural Gas algorithm for a Vector Quantization approach.
AMB	Bersini, Birattari, and Bontempi (1998)	An improved memory-based regression (MB) method (Platt, 1991) that uses an adaptive approach to automatically select the number of regressors (AMB).

Table 7.1: Summary of previous approaches for the prediction of the Mackey-Glass time series.

state (Bakker & Kleij, 2000). In tasks we considered so far this required remembering one or two events from the past, then using this information before over-writing the same memory cells. The Mackey-Glass equation, contains the input from $t-17$, hence its implementation requires the storage of all inputs from $t-17$ to t (time window approaches consider selected inputs back to at least $t-18$). Assuming that any dynamic model needs the event from time $t-\tau$ with $\tau \approx 17$, we note that the AR-RNN has to store all inputs from $t-\tau$ to t and to overwrite them at the adequate time. This requires the implementation of a circular buffer, a structure quite difficult for an RNN to simulate. In a TDNN, on the other hand, a circular buffer is inherent to the network structure.

7.3.3 Analysis

It is interesting that for MLPs ($T=6$) it was more effective to transform the task into a one-step-ahead prediction task and iterate than it was to predict directly (compare the results for $p=1$ and $p=T$). It is in general easier to predict fewer steps ahead, the disadvantage being that during iteration input values have to be replaced by predictions. For $T=6$ with $p=1$ this affects only the latest value. This advantage is lost for $T=84$ and the results with $p=1$ are worse than with $p=6$, where fewer iterations are necessary. For MLPs, iterated training did not in general produce better results: it improved performance when the step-size p was 1, and

Reference	Units	Para.	Seq.	NMSE		
				$T=1$	$T=6$	$T=84$
Predict Input: $x(t+T)=x(t)$		-	-	0.1466	0.8219	1.4485
Linear Predictor	-	-	-	0.0327	0.7173	1.5035
6th-order Polynom. (Crowder, 1990)	-	-	-	-	0.04	0.85
BPNN (Lapedes & Farber, 1987)	-	-	-	-	0.02	0.06
FTNN (Day & Davenport, 1993)	20	120	$7 \cdot 10^7$	-	0.012	-
ATNN (Day & Davenport, 1993)	20	120	$7 \cdot 10^7$	-	0.005	-
Cascade-Correlation (Crowder, 1990)	20	≈ 250	-		0.04	0.17
DCS-LLM (Chudy & Farkas, 1998)	200	200^2	$\approx 1 \cdot 10^5$	-	0.0055	0.03
EBPTTRNN (R. Bone et al., 2000)	6	65	-	-	0.0115	-
BGALR (Falco et al., 1998")	16	≈ 150	-	-	0.2373	0.267
EPNet (Yao & Liu, 1997)	≈ 10	≈ 100	$\approx 1 \cdot 10^4$	-	0.02	0.06
SOM	-	10x10	$\approx 1.5 \cdot 10^4$	-	0.013	0.06
(Vesanto, 1997)	-	35x35	$\approx 1.5 \cdot 10^4$	-	0.0048	0.022
Neural Gas (Martinez et al., 1993)	400	3600	$2 \cdot 10^4$	-	-	0.05
AMB (Bersini et al., 1998)	-		-	-	-	0.054
MLP, $p=T$	4	25	$1 \cdot 10^4$	0.0102	0.0511	0.4604
MLP, $p=T$	16	97	$1 \cdot 10^4$	0.0113	0.0502	0.4612
MLP, $p=1$	4	25	$1 \cdot 10^4$	$p=T=1$	0.0241	0.4208
MLP, $p=1$	16	97	$1 \cdot 10^4$	$p=T=1$	0.0252	0.4734
MLP, $p=1$, IT	4	25	$1 \cdot 10^4$	0.0089	0.0191	0.4143
MLP, $p=1$, IT	16	97	$1 \cdot 10^4$	0.0094	0.0205	0.3929
MLP, $p=6$	4	25	$1 \cdot 10^4$	-	$p=T=6$	0.1659
MLP, $p=6$	16	97	$1 \cdot 10^4$	-	$p=T=6$	0.1466
MLP, $p=6$, IT	4	25	$1 \cdot 10^4$	-	0.0946	0.3012
MLP, $p=6$, IT	16	97	$1 \cdot 10^4$	-	0.0945	0.2820
LSTM, $p=T$	4	113	$5 \cdot 10^4$	0.0214	0.1184	0.4700
LSTM, $p=1$	4	113	$5 \cdot 10^4$	$p=T=1$	0.1981	0.5927
LSTM, $p=1$, IT	4	113	$1 \cdot 10^4$	s. text	0.1970	0.8157
LSTM, $p=6$	4	113	$5 \cdot 10^4$	-	$p=T=6$	0.2910
LSTM, $p=6$, IT	4	113	$1 \cdot 10^4$	-	0.1903	0.3595

Table 7.2: Results for the Mackey-Glass task, showing (from left to right) the number of units, the number of parameters (weights for NNs), the number of training sequence presentations, and the NRMSE for prediction offsets $T \in \{1, 6, 84\}$. "IT" stands of iterated training.

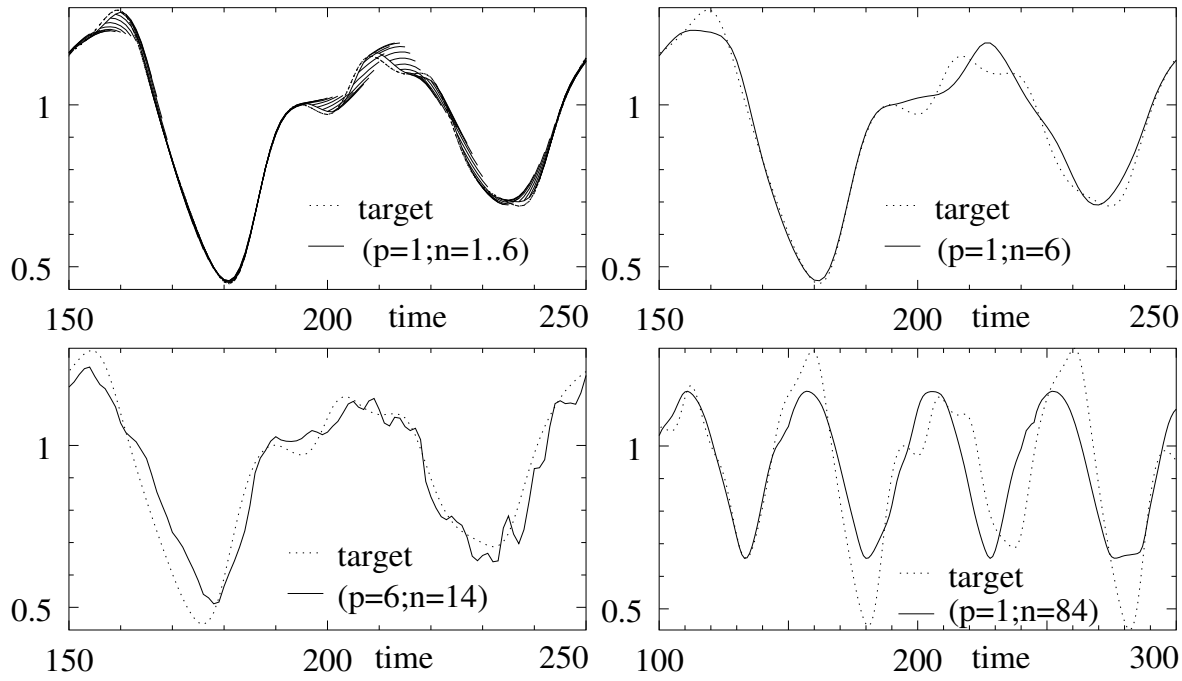

Figure 7.3: Mackey-Glass time series: Test run with LSTM network solutions. Shown are the network output as solid lines, and the target t. Top-Left: Single step prediction and six iterations ($p=1$, $T=1$, $n=1\ldots6$) after iterated training. Top-Right: The prediction for $T=6$ with $n=6$, extracted from the top-left graph. Bottom-Left: The best solution for $T=84$ with $p=6$ and $n=14$. Bottom-Left: The best single-step solution for $T=84$ with $p=1$ and $n=84$.

worsened performance for $p=6$.

The results for AR-LSTM approach are clearly worse than the results for time window approaches, for example with MLPs. Iterated training decreased the performance. But surprisingly, the relative performance decrease for one-step prediction was much larger than for iterated prediction. This indicates that the iteration capabilities were improved (taking in consideration the over-proportionally worsened one-step prediction performance).

The single-step predictions for LSTM are not accurate enough to follow the series for as much as 84 steps (Figure 7.3). Instead the LSTM network starts oscillating, having adapted to the strongest eigen-frequency in the task. During self-iterations, the memory cells tune into this eigen-oscillation (Figure 7.4), with time constants determined by the interaction of cell state and forget gate. Most solutions are stable during iterated testing as in the solution shown in Figure 7.4. Applying a sigmoid squashing function g prevents exponentially growth of the cell states by limiting their self-reinforcement to be linear. Linear self-reinforcement in turn can be compensated for by the forget gate. Still it is possible that networks diverge, when the damping induced by the forget gate is always smaller than the constant reinforcement described above. This situation might be established via feed back form the cell to the forget gate.

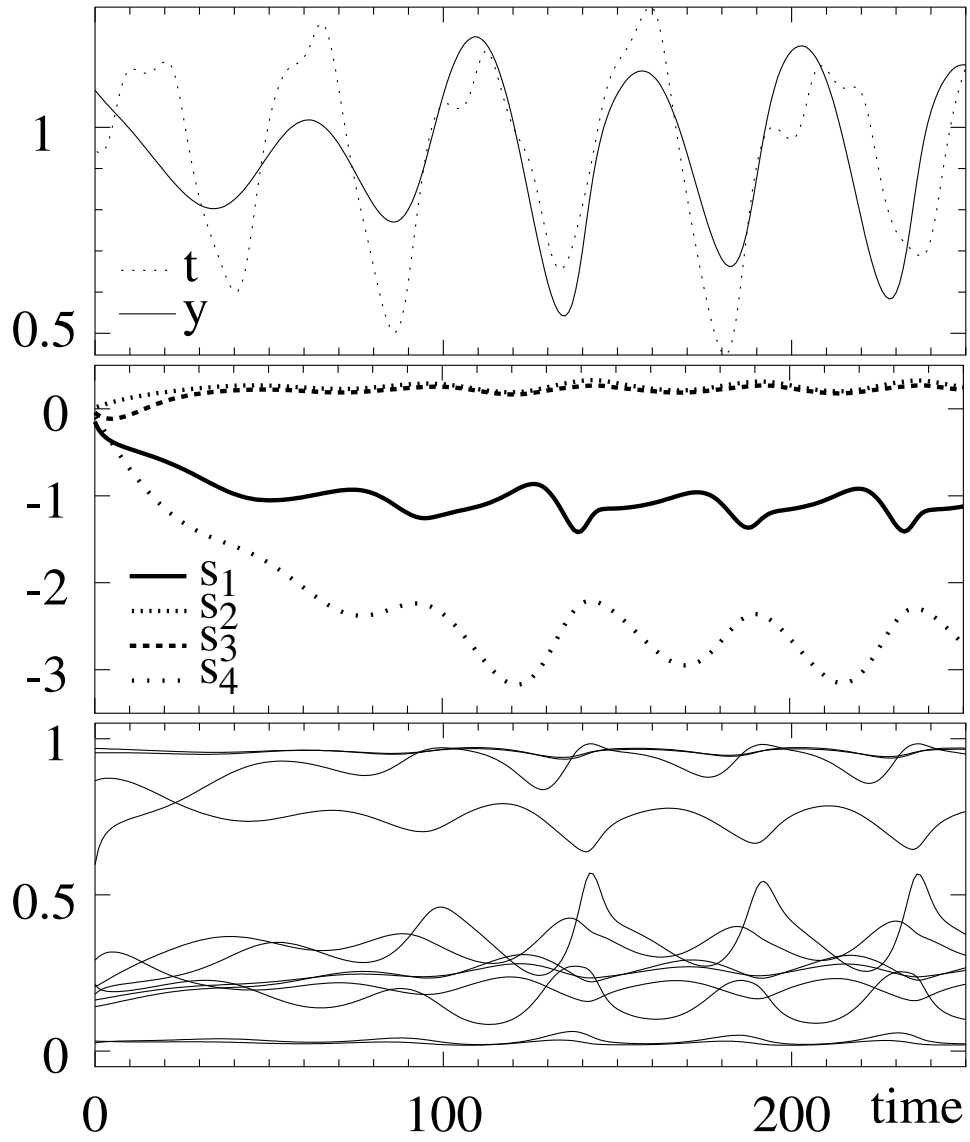

Figure 7.4: Test run with network solutions for the Mackey-Glass time series ($p = 1$, $T = 84$, $n = 84$). Shown is a "free" iteration of 250 steps starting with all states set to zero. Top: Network output y and the test set target t. Middle: Cell states s_c. Bottom: Activations of the gates.

7.4 Laser Data

This data is set A from the Santa Fe time series prediction competition (Weigend & Gershenfeld, 1993")[1]. It consists of one-dimensional data recorded from a Far-Infrared (FIR) laser in a chaotic state (Huebner, Abraham, & Weiss, 1989). The training set consists of 1,000 points from the laser, with the task being to predict the next 100 points (Figure 7.5). The main difficulty is to predict the collapse of activation in the test set, given only two similar events in the training set. We run tests for stepwise prediction and fully iterated prediction, where the output is clamped to the input for 100 steps.

[1]The data is available from http://www.stern.nyu.edu/~aweigend/Time-Series/SantaFe.html.

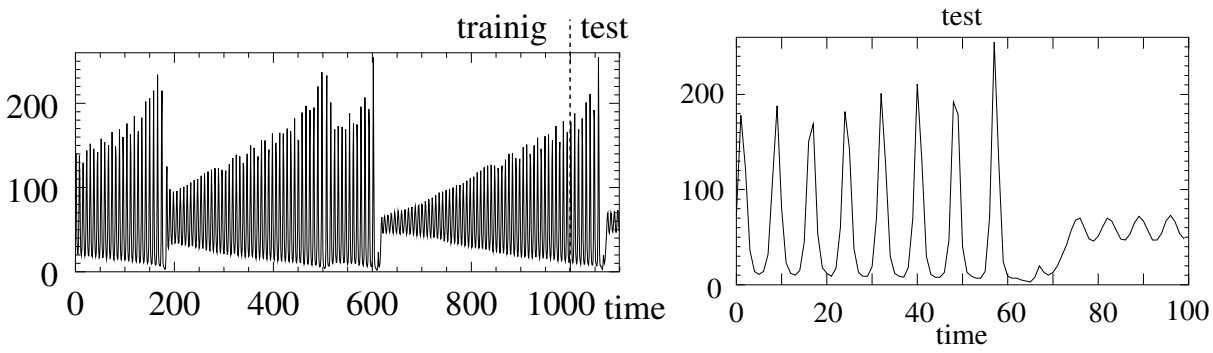

Figure 7.5: FIR-laser Data (Set A) from the Santa Fe time series prediction competition (see text for details).

For the experiments with MLPs the setup was as described for the Mackey-Glass data but with an input embedding of the last 9 time steps as in Koskela, Varsta, Heikkonen, and Kaski (1998).

7.4.1 Previous Work

Results are listed in Table 7.3. Linear prediction is no better than predicting the data-mean. Wan (1994) achieved the best results submitted to the original Santa Fe contest. He used a Finite Input Response Network (FIRN) (25 inputs and 12 hidden units), a method similar to a TDNN. Wan improved performance by replacing the last 25 predicted points by smoothed values (sFIRN).

Koskela, Varsta, Heikkonen, and Kaski (1998) compared recurrent SOMs (RSOMs) and MLPs (trained with the Levenberg-Marquardt algorithm) with an input embedding of dimension 9 (an input window with the last 9 values). Bakker, Schouten, Giles, Takens, and Bleek (2000) used a mixture of predictions and true values as input (Error Propagation, EP). Then Principal Component Analysis (PCA) was applied to reduce the dimensionality of the time embedding for the input from the 40 most recent inputs to 16 principal components. These were fed into a MLP (with two hidden layers of 32 and 24 units) and trained with BPTT using conjugate gradients. The value for the iterated prediction was achieved with a mixture of 90% clamped output and 10% true value (true iteration corresponds to 100% clamped output). The value for the iterated prediction was achieved without applying EP during training.

Kohlmorgen and Müller (1998) pointed out that the prediction problem could be solved by pattern matching, if it can be guaranteed that the best match from the past is always the right one. To resolve ambiguities they propose to up-sample the data using linear extrapolation (as done by Sauer, 1994).

The best result to date, according to our knowledge, was achieved by Weigend and Nix (1994). They used a nonlinear regression approach in a maximum likelihood framework, realized with feed-forward NN (25 inputs and 12 hidden units) using an additional output to estimate the prediction error. For the iterated prediction the mean of the values at times 620–700 was used as prediction after the predicted collapse of activity at time-step 1072 (this was based on visual inspection). A similar approach was used by Eric J. Kostelich (1994), who searched for the best match to an embedding of 75 steps using a local linear model.

McNames (2000) proposed a statistical method that used cross-validation error to estimate the model parameters for local models, but the testing conditions were too different to include

Reference	Units	Para.	Seq.	NMSE	
				stepwise	iterated
Predict Input: $x(t+T)=x(t)$	-	-	-	0.96836	-
Linear Predictor	-	-	-	1.25056	-
FIRN (Wan, 1994)	26	≈ 170	-	0.0230	0.0551
sFIRN (Wan, 1994)	26	≈ 170	-	-	0.0273
MLP (Koskela et al., 1998)	70	≈ 30	-	0.01777	-
RSOM (Koskela et al., 1998)	13	-	-	0.0833	-
EP-MLP (Bakker et al., 2000)	73	>1300	-	-	0.2159
(Sauer, 1994)	-	32	-	-	0.077
(Weigend & Nix, 1994)	27	≈ 180	-	0.0198	0.016
(Bontempi G., 1999)	-	-	-	-	0.029
MLP	16	177	$1 \cdot 10^4$	0.36322	>1
MLP	32	353	$1 \cdot 10^4$	0.0996017	0.856932
MLP	64	769	$1 \cdot 10^4$	0.101023	>1
MLP IT	32	353	$1 \cdot 10^4$	0.158298	0.621936
LSTM	4	113	$1 \cdot 10^5$	0.395959	1.02102
LSTM IT	4	113	$1 \cdot 10^5$	0.36422	0.96834

Table 7.3: Results for the FIR-laser task, showing (from left to right): The number of units, the number of parameters (weights for NNs), the number of training sequence presentations, and the NRMSE.

the results in the comparison. Bontempi G. (1999) used a similar approach called " Predicted Sum of Squares (PRESS)" (here, the dimension of the time embedding was 16).

7.4.2 Results

The results for MLP and LSTM are listed in Table 7.3. The results for these methods are not as good as the other results listed in Table 7.3. This is true in part because we did not replace predicted values by hand with a mean value where we suspected the system to be lead astray.

7.4.3 Analysis

The LSTM network could not predict the collapse of emission in the test set (Figure 7.6). Instead, the network tracks the oscillation in the original series for only about 40 steps before desynchronizing. This indicates performance similar to that in the Mackey-Glass task: the LSTM network was able to track the strongest eigen-frequency in the task but was unable to account for high-frequency variance. Though the MLP performed better, it generated inaccurate amplitudes and also desynchronized after about 40 steps. The MLP did however manage to predict the collapse of emission (Figure 7.6).

LSTM's ability to track slow oscillations in the chaotic signal is notable. In simple cases, synchronization with a periodic signal is easily achieved using mechanisms such as phase-locked loops (PLLs). But when noisy or complex signals are used, synchronization can be challenging

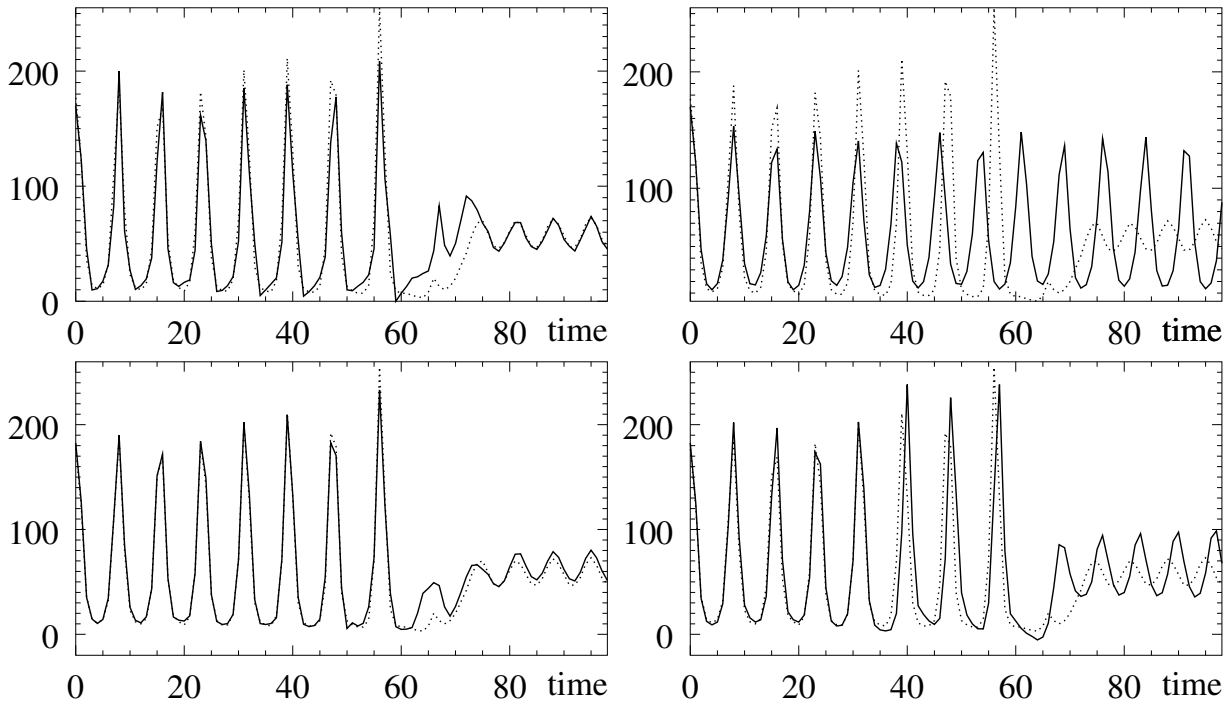

Figure 7.6: Test run with network solutions after iterated training for the FIR-laser task. Top: LSTM. Bottom: MLP with 32 hidden units. Left: Single-Step prediction. Right: Iteration of 100 steps.

(McAuley, 1994; Large & Kolen, 1994). Systems like LSTM that can find periodicity in complicated signals should be applicable to cognitive domains such as speech and music (Large & Jones, 1999; Eck, 2000a). See also (Eck, 2000b) for more on this topic.

Iterated training yielded improved results for iterated prediction, even when stepwise prediction made things worse, as in the case of MLP single-step prediction (prediction step size one) for both the Mackey-Glass task and the FIR task. When multi-step prediction was used (for Mackey-Glass only), iterated training did not improve system performance.

7.5 Conclusion

A time window based MLP outperformed the LSTM pure-AR approach on certain time series prediction benchmarks solvable by looking at a few recent inputs only. Thus LSTM's special strength, namely, to learn to remember single events for very long, unknown time periods, was not necessary here.

LSTM learned to tune into the fundamental oscillation of each series but was unable to accurately follow the signal. The MLP, on the other hand, was able to capture some aspects of the chaotic behavior. For example the system could predict the collapse of emission in the FIR-laser task.

Iterated training has advantages over single-step training for iterated testing only for MLPs and when the prediction step-size is one. The advantage is evident when the number of necessary iterations is large.

Our results suggest to use LSTM only on tasks where traditional time window based ap-

proaches must fail. One reasonable *hybrid* approach to prediction of unknown time series may be this: start by training a time window-based MLP, then freeze its weights and use LSTM only to reduce the residual error if there is any, employing LSTM's ability to cope with long time lags between significant events.

LSTM's ability to track slow oscillations in the chaotic signal may be applicable to cognitive domains such as rhythm detection in speech and music.

Chapter 8

Conclusion

This work has concentrated on improving and applying the original LSTM algorithm as introduced by Hochreiter and Schmidhuber (1997). We proposed to extended LSTM with forget gates and peephole connections. Extended LSTM is clearly superior to traditional LSTM (and other RNNs), and can serve as basis for future applications. Our findings reinforce the perception that LSTM is a very general and promising adaptive sequence processing device, with a wider field of potential applications than alternative RNNs. In the following we summarize the contributions of this thesis and present some thoughts about future work and possible LSTM applications.

8.1 Main Contributions

Forget Gates. While previous work focused on training sequences with well-defined beginnings and ends, 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.

Forget gates also substantially improve LSTM's performance on tasks involving arithmetic operations, because they make the LSTM architecture more powerful.

Extending LSTM with peephole connections. We identified a weakness in the wiring scheme of the multiplicative gates surrounding LSTM's constant error carrousels (CECs). As a remedy, we extend LSTM by introducing peephole connections from the CECs to the gates, that allow them to inspect the current internal cell-states.

Timing. We tested LSTM on a special class of tasks that requires the network to extract relevant information conveyed by the duration of intervals between events. We showed that LSTM can solve such highly nonlinear tasks as well, by learning to precisely measure time intervals, provided we furnish LSTM cells with peephole connections.

Context free and context sensitive languages. We show that LSTM outperforms other

RNNs on context free language (CFL) benchmarks. Moreover, LSTM is the first RNN to learn a context sensitive language.

Time series prediction. Time window based MLPs outperformed a LSTM pure auto-regressive approach on certain time series prediction benchmarks solvable by looking at a few recent inputs only. Thus LSTM's special strength, namely, to learn to remember single events for very long, unknown time periods, was not necessary for those tasks.

8.2 Future work and possible applications of LSTM.

Gain adaptation. In our experiments we either used a constant learning rate (sometimes with exponential or linear decay within sequences) or applied the rather simple momentum algorithm (Plaut et al., 1986). More advanced local learning rate adaptation approaches like a decoupled Kalman filtering (Puskorius & Feldkamp, 1994) or stochastic meta descent (Schraudolph, 1999, 2000) may improve learning speed and reduce the percentage of networks that diverge.

Hierarchical decomposition, rhythm and timing. LSTM with forget gates holds promise for any sequential processing task in which we suspect that a hierarchical decomposition may exist, but do not know in advance what this decomposition is (one example is prosodic information in speech). We showed that memory blocks equipped with forget gates and peephole connections are capable of developing into internal oscillators and timers and that LSTM is able to track slow oscillations in the chaotic signal. This may allow the recognition and generation of hierarchical rhythmic patterns in music. In particular the ability to perform precise timing and measuring makes LSTM a promising approach for real-world tasks whose solution partly depend on the precise duration of intervals between relevant events.

Growing LSTM networks. It may be useful to grow LSTM networks (e.g., add one memory block at a time), similar to the cascade-correlation algorithm (Fahlman, 1991), to decouple blocks when tracking multiple frequencies in a signal. So far only the fundamental frequency was tracked.

Time series prediction. For the prediction of unknown time series our results suggest to use LSTM in a *hybrid* approach as follows: start by training a time window-based MLP, then freeze its weights and use LSTM only to reduce the residual error if there is any, employing LSTM's ability to cope with long time lags between significant events. An example for a task where a hybrid approach with LSTM might be promising is the prediction of secondary protein structure from a sequence of amino acids (Brunak, Baldi, Frasconi, Pollastri, & Soda, 1999). The standard solution involves using a fixed window over the protein sequence, centered over a specific amino acid. As a protein is folded, acids that are far apart in the series of acids may be spatially close and have significant interaction. This generates complex, varying long-term dependencies in the series.

Appendix A

Embedded Reber Grammar Statistics

The minimal length of an embedded Reber grammar (ERG) string is 9; string length have no upper bound. To provide an idea of the string size distribution, Figure A.1 (left) shows a histogram of ERG strings computed from sampled data. We assume that ERG string probabilities decrease exponentially with ERG string size (compare exponential fit on the left hand side of Figure A.1), so that the probability $p(l)$ of sampling a string of size l can be written as:

$$p(l) = b\ \mathrm{e}^{-a(l-9)} \quad \text{for} \quad l \geq 9 \quad \text{else } p(l) = 0 \quad ,$$

with $a, b > 0$; the offset 9 expresses the minimum string length. To compute the probability $P(L)$ of sampling a string of size $l \leq L$ we integrate $p(l)$:

$$P(L) = \int_o^L p(l)\ dl = \frac{b}{a}\left(1 - \mathrm{e}^{-a(L-9)}\right) \quad .$$

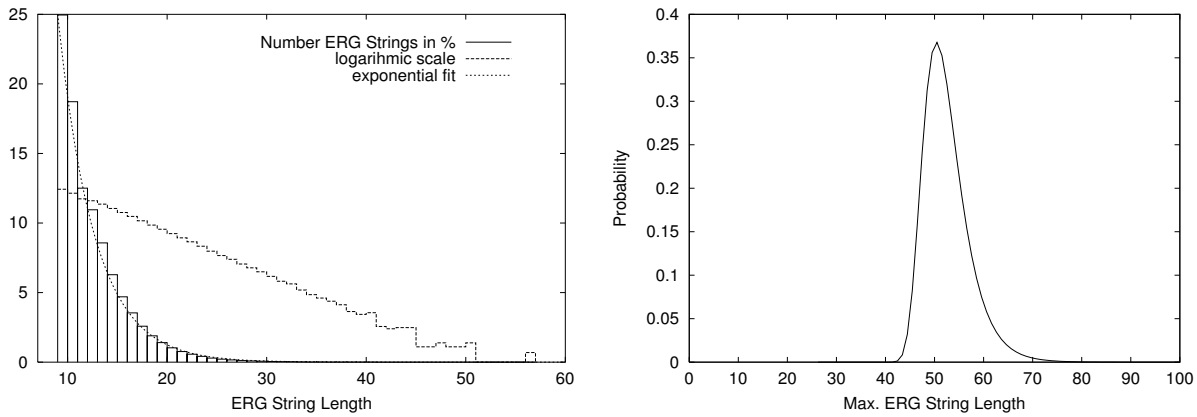

Figure A.1: Left: histogram of 10^6 random samples of ERG string sizes. Right: Joint probability that an ERG string of a given size occurs and is the longest among 80000.

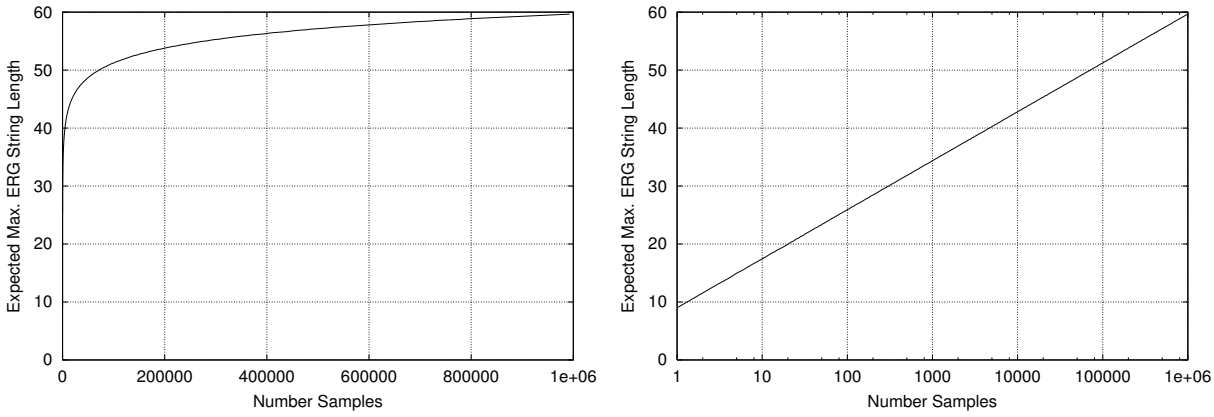

Figure A.2: Left: number of embedded Reber strings N plotted against lower bounds of expected maximal string size $\bar{\tau}(N)$. Right: logarithmic x-axis.

From normalization $P(\infty) \stackrel{!}{=} 1$ follows $a = b$. Solving $P(\bar{l}) \stackrel{!}{=} 1 - P(\bar{l})$ with the value $a \approx \frac{3}{11}$ extracted form the data (left hand side of Figure A.1), we find the expected ERG string size:

$$\bar{l} = o - \frac{1}{a}\ln(\frac{1}{2}) \approx 11.54 \quad . \tag{A.1}$$

Given a set of N ERG strings, what is the expected maximal string length $\bar{\tau}(N)$? We derive a lower bound $P_N(\tau)$ for the probability that a set of N ERG strings contains a string of size $\geq \tau$, assuming a sample of $N-1$ strings of size $\leq \tau$ and one of size $\geq \tau$ (we set $N - 1 \approx N$):

$$P_N(\tau) = N \cdot P(\tau)^N \cdot (1 - P(\tau)) \quad .$$

Figure A.1 (right) plots P_N for $N = 80000$. The x-value of the distribution maximum is a lower bound for $\bar{\tau}(N = 80000)$. Figure A.2 plots N against the lower bound of $\bar{\tau}(N)$.

$\bar{\tau}(N)$ grows logarithmically with N. For the test set we use in our experiments ($N = 80000$) the expected maximal string length is about 50.

Appendix B

Peephole LSTM with Forget Gates in Pseudo-code

The pseudo-code in this chapter describes the implementation of LSTM with forget gates and peephole connections as introduced in the chapters 3 and 5. This is the LSTM version that we currently use and recommend; the C code can be down-loaded from: "http://www.idsia.ch/~felix".

The partial derivatives $\frac{\partial s}{\partial w}$ are represented by the variables dS:

$$dS_{l_j m}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{l_j m}} \quad ;$$

as defined in chapter 2, j indexes memory blocks and v indexes memory cells in block j; $l = c_j^v$ for weights to the cell, $l = in$ for weights to the input gate, and $l = \varphi$ for weights to the forget gate. The variables dS are calculated no matter if a target (and hence an error) is given or not. Thus their calculation is done in the forward pass. Whereas the backward pass is only calculated at time steps when a target is present.

It is task-specific (see descriptions in chapters) when the weight-updates are executed: After each step time, regularly after a fixed number of time steps, after intervals with varying duration or at the and of a sequence or epoch.

The momentum algorithm (Plaut et al., 1986), that we used for some of our experiments, is not incorporated into this pseudo-code.

init network:

reset: CECs: $s_{c_j^v} = \hat{s}_{c_j^v} = 0$; **partials:** $dS = 0$; **activations:** $y = \hat{y} = 0$;

forward pass:

input units: $y =$ `current external input;`

roll over: activations: $\hat{y} = y$; **cell states:** $\hat{s}_{c_j^v} = s_{c_j^v}$;

`loop over memory blocks, indexed` j `{`

Step 1a: input gates (5.1):

$net_{in_j} = \sum_m w_{in_j m}\ \hat{y}^m + \sum_{v=1}^{S_j} w_{in_j c_j^v}\ \hat{s}_{c_j^v}$; $\quad y^{in_j} = f_{in_j}(net_{in_j})$;

Step 1b: forget gates (5.2):

$net_{\varphi_j} = \sum_m w_{\varphi_j m}\ \hat{y}^m + \sum_{v=1}^{S_j} w_{\varphi_j c_j^v}\ \hat{s}_{c_j^v}$; $\quad y^{\varphi_j} = f_{\varphi_j}(net_{\varphi_j})$;

Step 1c: CECs, i.e the cell states (5.3):

`loop over the` S_j `cells in block` j`, indexed` v `{`

$net_{c_j^v} = \sum_m w_{c_j^v m}\ \hat{y}^m$; $\quad s_{c_j^v} = y^{\varphi_j}\ \hat{s}_{c_j^v} + y^{in_j}\ g(net_{c_j^v})$; `}`

Step 2:

output gate activation: (5.4):

$net_{out_j} = \sum_m w_{out_j m}\ \hat{y}^m + \sum_{v=1}^{S_j} w_{out_j c_j^v}\ s_{c_j^v}$; $\quad y^{out_j} = f_{out_j}(net_{out_j})$;

cell outputs (5.5):

`loop over the` S_j `cells in block` j`, indexed` v `{` $y^{c_j^v} = y^{out_j}\ s_{c_j^v}$; `}`

`} end loop over memory blocks`

output units (2.9): $net_k = \sum_m w_{km}\ y^m$; $\quad y^k = f_k(net_k)$;

partial derivatives:

`loop over memory blocks, indexed` j `{`

`loop over the` S_j `cells in block` j`, indexed` v `{`

cells (5.6), ($dS_{cm}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{c_j^v m}}$):

$dS_{cm}^{jv} = dS_{cm}^{jv}\ y^{\varphi_j} + g'(net_{c_j^v})\ y^{in_j}\ \hat{y}^m$;

input gates (5.7), (5.7b), ($dS_{in,m}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{in_j m}}$, $dS_{in,c_j^{v'}}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{in_j c_j^{v'}}}$):

$dS_{in,m}^{jv} = dS_{in,m}^{jv}\ y^{\varphi_j} + g(net_{c_j^v})\ f'_{in_j}(net_{in_j})\ \hat{y}^m$;

`loop over peephole connections from all cells, indexed` v' `{`

$dS_{in,c_j^{v'}}^{jv} = dS_{in,c_j^{v'}}^{jv}\ y^{\varphi_j} + g(net_{c_j^v})\ f'_{in_j}(net_{in_j})\ \hat{s}_c^{v'}$; `}`

forget gates (5.8), (5.8b), ($dS_{\varphi m}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\varphi_j m}}$, $dS_{\varphi c_j^{v'}}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\varphi_j c_j^{v'}}}$):

$dS_{\varphi m}^{jv} = dS_{\varphi m}^{jv}\ y^{\varphi_j} + \hat{s}_{c_j^v}\ f'_{\varphi_j}(net_{\varphi_j})\ \hat{y}^m$;

`loop over peephole connections from all cells, indexed` v' `{`

$dS_{\varphi c_j^{v'}}^{jv} = dS_{\varphi c_j^{v'}}^{jv}\ y^{\varphi_j} + \hat{s}_{c_j^v}\ f'_{\varphi_j}(net_{\varphi_j})\ \hat{s}_c^{v'}$; `}`

`}` `} end loops over cells and memory blocks`

backward pass (if error injected):

errors and δs:

injection error: $e_k = t^k - y^k$;

δs of output units (5.10): $\delta_k = f'_k(net_k)\ e_k$;

loop over memory blocks, indexed j {

δs of output gates (5.11b):

$\delta_{out_j} = f'_{out_j}(net_{out_j})\ \left(\sum_{v=1}^{S_j}\ s_{c_j^v} \sum_k w_{kc_j^v}\ \delta_k\right)$;

internal state error (5.15):

loop over the S_j cells in block j, indexed v {

$e_{s_{c_j^v}} = y^{out_j}\ \left(\sum_k w_{kc_j^v}\ \delta_k\right)$; }

} end loop over memory blocks

weight updates:

output units (5.9): $\Delta w_{km} = \alpha\ \delta_k\ y^m$;

loop over memory blocks, indexed j {

output gates (5.11a):

$\Delta w_{out,m} = \alpha\ \delta_{out}\ \hat{y}^m$; $\Delta w_{out,c_j^v} = \alpha\ \delta_{out}\ s_{c_j^v}$;

input gates (5.13):

$\Delta w_{in,m} = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}\ dS^{jv}_{in,m}$;

loop over peephole connections from all cells, indexed v' {

$\Delta w_{in,c_j^{v'}} = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}\ dS^{jv}_{in,c_j^{v'}}$; }

forget gates (5.14):

$\Delta w_{\varphi m} = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}\ dS^{jv}_{\varphi m}$;

loop over peephole connections from all cells, indexed v' {

$\Delta w_{\varphi c_j^{v'}} = \alpha \sum_{v=1}^{S_j}\ e_{s_{c_j^v}}\ dS^{jv}_{\varphi c_j^{v'}}$; }

cells (5.12):

loop over the S_j cells in block j, indexed v {

$\Delta w_{c_j^v m} = \alpha\ e_{s_{c_j^v}}\ dS^{jv}_{cm}$; };

} end loop over memory blocks

References

Bakker, B., & Kleij, G. van der Voort van der. (2000). Trading off perception with internal state: Reinforcement learning and analysis of q-elman networks in a markovian task. In *Proceedings of IJCNN 2000.* Como, Italy.

Bakker, R., Schouten, J. C., Giles, C. L., Takens, F., & Bleek, C. M. van den. (2000). Learning chaotic attractors by neural networks. *Neural Computation*, *12*(10).

Bengio, Y., & Frasconi, P. (1995). An input output HMM architecture. In *Advances in Neural Information Processing Systems 7.* San Mateo CA: Morgan Kaufmann.

Bengio, Y., Frasconi, P., Gori, M., & G.Soda. (1993). Recurrent neural networks for adaptive temporal processing. In *Proceedings of the 6th italian workshop on parallel architectures and neural networks wirn93* (pp. 85–117). Vietri (Italy): World Scientific Pub.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157-166.

Bersini, H., Birattari, M., & Bontempi, G. (1998). Adaptive memory-based regression methods. In *In Proceedings of the 1998 IEEE International Joint Conference on Neural Networks* (pp. 2102–2106).

Blair, A. D., & Pollack, J. B. (1997). Analysis of dynamical recognizers. *Neural Computation*, *9*(5), 1127–1142.

Bontempi G., B. H., Birattari M. (1999). Local learning for iterated time-series prediction. In B. I. & D. S. (Eds.), *Machine Learning: Proceedings of the Sixteenth International Conference* (p. 32-38). San Francisco, USA: Morgan Kaufmann.

Box, G., & Jenkins, G. (1970"). *Time series analysis – forecasting and control; san francisco: Holden-day.*

Brunak, S., Baldi, P., Frasconi, P., Pollastri, G., & Soda, G. (1999). Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, *15*(11).

Casey, M. P. (1996). The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, *8*(6), 1135–1178.

Chudy, L., & Farkas, I. (1998). Prediction of chaotic time-series using dynamic cell structuresand local linear models. *Neural Network World*, *8*(5), 481-489.

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

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In J. Grefenstette (Ed.), *Proceedings of an international conference on genetic algorithms and their applications.* Hillsdale NJ: Lawrence Erlbaum Associates.

Crowder, R. S. (1990). Predicting the mackey-glass timeseries with cascadecorrelation learning. In D. S. T. (ed) (Ed.), *Connectionist Models: Proceedings of the 1990 Summer School.*

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

Darken, C. (1995). Stochastic approximation and neural network learning. In M. A. Arbib (Ed.), *The Handbook of Brain Theory and Neural Networks* (pp. 941–944). Cambridge, Massachusetts: MIT Press.

Das, S., Giles, C., & Sun, G. (1992). Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proceedings of The Fourteenth Annual Conference of the Cognitive Science Society* (pp. 791–795). San Mateo, CA: Morgan Kaufmann Publishers.

Day, S. P., & Davenport, M. R. (1993). Continuous-time temporal back-progagation with adaptive time delays. *IEEE Transactions on Neural Networks, 4*, 348–354.

Deco, G., & Schürmann, B. (1994). Neural learning of chaotic system behavior. *IEICE Trans. Fundamentals, E77-A*, 1840-1845.

Dickmanns, D., Schmidhuber, J., & Winklhofer, A. (1987). *Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.*

Doya, K., & Yoshizawa, S. (1989). Adaptive neural oscillator using continuous-time backpropagation learning. *Neural Networks, 2*(5), 375–385.

Eck, D. (2000a). *Meter Through Synchrony: Processing Rhythmical Patterns with Relaxation Oscillators.* Unpublished doctoral dissertation, Indiana University, Bloomington, IN., (www.idsia.ch/∼doug/publications.html).

Eck, D. (2000b). *Tracking rhythms with a relaxation oscillator* (Tech. Rep. No. IDSIA-10-00). www.idsia.ch/techrep.html, Galleria 2, 6928 Manno-Lugano, Switzerland: IDSIA.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science, 14*(2), 179–211.

Eric J. Kostelich, D. P. L. (1994). The prediction of chaotic time series: a variation on the method of analogues. In W. A. S. & G. N. A. (Eds.), *Time Series Prediction: Forecasting the Future and Understanding the Past* (pp. 283–295). Addison-Wesley.

Fahlman, S. E. (1991). The recurrent cascade-correlation learning algorithm. In R. P. Lippmann, J. E. Moody, & D. S. Touretzky (Eds.), *NIPS 3* (p. 190-196). San Mateo, CA: Morgan Kaufmann.

Falco, I. de, Iazzetta, A., Natale, P., & Tarantino, E. (1998"). Evolutionary neural networks for nonlinear dynamics modeling. In *Parallel Problem Solving from Nature 98* (Vol. 1498, p. 593-602). Springer.

Forcada, M. L., & Carrasco, R. C. (1995). Learning the initial state of a second-order recurrent neural network during regular-language inference [Letter]. *Neural Computation, 7*(5), 923–930.

Gers, F. A., Eck, D., & Schmidhuber, J. (2000). *Applying LSTM to time series predictable through time-window approaches* (Tech. Rep. No. IDSIA-22-00). Manno, CH: IDSIA.

Gers, F. A., Eck, D., & Schmidhuber, J. (2001a). Applying LSTM to time series predictable through time-window approaches. In *Proc. ICANN 2001, Int. Conf. on Artificial Neural Networks.* Vienna, Austria: IEE, London. (submitted)

Gers, F. A., Eck, D., & Schmidhuber, J. (2001b). Applying LSTM to time series predictable through time-window approaches. In *Neural nets, WIRN vietri-99, proceedings 11th workshop on neural nets.* Vietri sul Mare, Italy. (submitted)

Gers, F. A., & Schmidhuber, J. (2000a). Neural processing of complex continual input streams. In *Proc. IJCNN'2000, Int. Joint Conf. on Neural Networks.* Como, Italy.

Gers, F. A., & Schmidhuber, J. (2000b). *Neural processing of complex continual input streams* (Tech. Rep. No. IDSIA-02-00). Manno, CH: IDSIA.

Gers, F. A., & Schmidhuber, J. (2000c). Recurrent nets that time and count. In *Proc. IJCNN'2000, Int. Joint Conf. on Neural Networks.* Como, Italy.

Gers, F. A., & Schmidhuber, J. (2000d). *Recurrent nets that time and count* (Tech. Rep. No. IDSIA-01-00). Manno, CH: IDSIA.

Gers, F. A., & Schmidhuber, J. (2000e). LSTM learns context free languages. In *Snowbird 2000 Conference.*

Gers, F. A., & Schmidhuber, J. (2000f). *Long Short-Term Memory learns context free languages and context sensitive languages* (Tech. Rep. No. IDSIA-03-00). Manno, CH: IDSIA.

Gers, F. A., & Schmidhuber, J. (2001a). LSTM recurrent networks learn simple context free and context sensitive languages. *IEEE Transactions on Neural Networks.* (accepted)

Gers, F. A., & Schmidhuber, J. (2001b). Long Short-Term Memory learns context free and context sensitive languages. In *Proceedings of the ICANNGA 2001 conference.* Springer. (accepted)

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999a). Continual prediction using LSTM with forget gates. In M. Marinaro & R. Tagliaferri (Eds.), *Neural Nets, WIRN Vietri-99, Proceedings 11th Workshop on Neural Nets* (p. 133-138). Vietri sul Mare, Italy: Springer Verlag, Berlin.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999b). Learning to forget: Continual prediction with LSTM. In *Proc. ICANN'99, Int. Conf. on Artificial Neural Networks* (Vol. 2, p. 850-855). Edinburgh, Scotland: IEE, London.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999c). *Learning to forget: Continual prediction with LSTM* (Tech. Rep. No. IDSIA-01-99). Lugano, CH: IDSIA.

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

Gers, F. A., Schmidhuber, J., & Schraudolph, N. Learning precise timing with LSTM recurrent networks. (submitted to Neural Computation)

Haffner, P., & Waibel, A. (1992). Multi-state time delay networks for continuous speech recognition. In J. E. Moody, S. J. Hanson, & R. P. Lippmann (Eds.), *Advances in Neural Information Processing Systems* (Vol. 4, pp. 135–142). Morgan Kaufmann Publishers, Inc.

Hinton, G. E., Sejnowski, T. J., & Ackley, D. H. (1984). *Boltzmann Machines: Constraint satisfaction networks that learn* (Tech. Rep. No. CMU-CS-84-119). Carnegie Mellon University.

Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.* (See www7.informatik.tu-muenchen.de/~hochreit)

Hochreiter, S., & Schmidhuber, J. (1995). Long short-term memory can solve hard long time lag problems. In G. Tesauro, D. S. Touretzky, & T. K. Leen (Eds.), *Advances in neural information processing systems 7 (NIPS "94).* Cambridge, MA: MIT Press.

Hochreiter, S., & Schmidhuber, J. (1996). Bridging long time lags by weight guessing and "Long Short-Term Memory". In F. L. Silva, J. C. Principe, & L. B. Almeida (Eds.), *Spatiotemporal models in biological and artificial systems* (p. 65-72). IOS Press, Amsterdam, Netherlands. (Serie: Frontiers in Artificial Intelligence and Applications, Volume 37)

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

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational facilities. *Proceedings of the National Academy of Sciences of the USA, 79*, 2554–2558.

Huebner, U., Abraham, N. B., & Weiss, C. O. (1989). Dimensions and entropies of chaotic intensity pulsations in a single-mode far-infrared nh3 laser. *Phys. Rev. A, 40*, 6354.

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Cognitive Science Society Conference.* Hillsdale, NJ: Erlbaum.

Kalinke, Y., & Lehmann, H. (1998). Computation in recurrent neural networks: From counters to iterated function systems. In G. Antoniou & J. Slaney (Eds.), *Advanced Topics in Artificial Intelligence, Proceedings of the 11th Australian Joint Conference on Artificial Intelligence* (Vol. 1502). Berlin,Heidelberg: Springer.

Kohlmorgen, J., & Müller, K.-R. (1998). Data set a is a pattern matching problem. *Neural Processing Letters, 7*(1), 43-47.

Koskela, T., Varsta, M., Heikkonen, J., & Kaski, K. (1998). Recurrent SOM with local linear models in time series prediction. In *6th European Symposium on Artificial Neural Networks. ESANN'98. Proceedings. D-Facto, Brussels, Belgium* (pp. 167–72).

Koza, J. R. (1992). *Genetic programming.* Cambridge, MA: MIT Press.

Lapedes, A., & Farber, R. (1987). *Nonlinear signal processing using neural networks: Prediction and signal modeling* (Tech. Rep. Nos. LA–UR–87–2662). Los Alamos, New Mexico: Los Alamos National Laboratory.

Large, E. W., & Jones, M. R. (1999). The dynamics of attending: How people track time-varying events. *Psychological Review, 106*(1), 119–159.

Large, E. W., & Kolen, J. F. (1994). Resonance and the perception of musical meter. *Connection Science, 6*, 177–208.

LeCun, Y., Bottou, L., Orr, G., & Müller, K.-R. (1998). Efficient backprop. In G. B. Orr & K.-R. Müller (Eds.), *Neural Networks—Tricks of the Trade* (Vol. 1524, p. 5-50). Berlin: Springer Verlag.

Lee, L. (1996). *Learning of context-free languages: A survey of the literature* (Tech. Rep. No. TR-12-96). Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts.

Lin, T., Horne, B. G., Tiño, P., & Giles, C. L. (1996). Learning long-term dependencies in NARX recurrent neural networks [Paper]. *IEEE Transactions on Neural Networks, 7*(6), 1329–1338.

Mackey, M., & Glass, L. (1977). Oscillation and chaos in a physiological control system. *Science, 197*(287).

Martinez, T. M., Berkovich, S. G., & Schulten, K. J. (1993). Neural-gas network for vector quantization and its application to time-series prediction [Paper]. *IEEE Transactions on Neural Networks, 4*(4), 558–569.

McAuley, J. (1994). Finding metrical structure in time. In M. Mozer, P. Smolensky, D. Touretsky, J. Elman, & A. S. Weigend (Eds.), *Proceedings of the 1993 Connectionist Models Summer School* (pp. 219–227). Hillsdale, NJ: Erlbaum.

McNames, J. (2000). Local modeling optimization for time series prediction. In *In Proceedings of the 8th European Symposium on Artificial Neural Networks* (p. 305-310). Bruges, Belgium.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*(63), 81-97.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1*, 47–80.

Mozer, M. C. (1989). A focused backpropagation algorithm for temporal pattern processing. *Complex Systems, 3*, 349–381.

Mozer, M. C. (1992). Induction of multiscale temporal structure. In D. S. Lippman, J. E. Moody, & D. S. Touretzky (Eds.), *Advances in Neural Information Processing Systems 4* (p. 275-282). San Mateo, CA: Morgan Kaufmann.

Mozer, M. C. (1993). Neural net architectures for temporal sequences processing. In A. S. Weigend & N. A. Gershenfeld (Eds.), *Time series prediction: Forecasting the future and understanding the past* (Vol. 15, pp. 243–264). Reading, MA: Addison Wesley.

Osborne, M., & Briscoe, E. (1997). Learning stochastic categorial grammars. In *Proceedings of the Assoc. for Comp. Linguistics, Comp. Nat. Lg.Learning (CoNLL97) Workshop* (pp. 80–87). Madrid. (http://citeseer.nj.nec.com/osborne97learning.html)

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

Platt, J. (1991). A resource-allocating network for function interpolation. *Neural Computation*, *3*, 213–225.

Plaut, D. C., Nowlan, S. J., & Hinton, G. E. (1986). *Experiments on learning back propagation* (Tech. Rep. Nos. CMU–CS–86–126). Pittsburgh, PA: Carnegie–Mellon University.

Porter, Bruce, W., Bareiss, R., & Holte, R. C. (1990). Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligence*, *45*(1-2), 229–263.

Principe, J. C., & Kuo, J.-M. (1995). Dynamic modelling of chaotic time series with neural networks. In G. Tesauro, D. Touretzky, & T. Leen (Eds.), *Advances in Neural Information Processing Systems* (Vol. 7, pp. 311–318). The MIT Press.

Principe, J. C., Rathie, A., & Kuo, J. M. (1992). Prediction of chaotic time series with neural networks and the issue of dynamic modeling. *Int. J. of Bifurcation and Chaos*, *2*(4), 989–996.

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.

Quinlan, J. (1992). *Programs for machine learning.* Morgan Kaufmann.

R. Bone, Crucianu, M., Verley, G., & Asselin de Beauville, J.-P. (2000). A bounded exploration approach to constructive algorithms for recurrent neural networks. In *Proceedings of IJCNN 2000.* Como, Italy.

Ring, M. B. (1994). *Continual learning in reinforcement environments.* Unpublished doctoral dissertation, University of Texas at Austin, Austin, Texas 78712.

Robinson, A. J., & Fallside, F. (1987). *The utility driven dynamic error propagation network* (Tech. Rep. No. CUED/F-INFENG/TR.1). Cambridge University Engineering Department.

Rodriguez, P., & Wiles, J. (1998). Recurrent neural networks can learn to implement symbol-sensitive counting. In *Advances in Neural Information Processing Systems* (Vol. 10, p. 87-93). The MIT Press.

Rodriguez, P., Wiles, J., & Elman, J. (1999). A recurrent neural network that learns to count. *Connection Science*, *11*(1), 5-40.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representation by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1, pp. 318–362). Cambridge, MA: MIT Press.

Sakakibara, Y. (1997). Recent advances of grammatical inference. *Theoretical Computer Science*, *185*(1), 15–45.

Sałustowicz, R. P., & Schmidhuber, J. (1997). Probabilistic incremental program evolution: Stochastic search through program space. In M. van Someren & G. Widmer (Eds.), *Machine Learning: ECML-97, Lecture Notes in Artificial Intelligence 1224* (p. 213-220). Springer-Verlag Berlin Heidelberg.

Sauer, T. (1994). Time series prediction using delay coordinate embedding. In A. S. Weigend & N. A. Gershenfeld (Eds.), *Time Series Prediction: Forecasting the Future and Understanding the Past.* Addison-Wesley.

Schmidhuber, J. (1989). The Neural Bucket Brigade, a local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, *1*(4), 403-412.

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

Schmidhuber, J., & Hochreiter, S. (1996). *Guessing can outperform many long time lag algorithms* (Tech. Rep. No. IDSIA-19-96). IDSIA.

Schraudolph, N. (1999). Local gain adaptation in stochastic gradient descent. In *Proceedings of the 9th International Conference on Artificial Neural Networks.* London: IEE.

Schraudolph, N. N. (2000). *Fast second-order gradient descent via O(n) curvature matrix-vector products* (Tech. Rep. No. IDSIA-12-00). Galleria 2, CH-6928 Manno, Switzerland: Istituto Dalle Molle di Studi sull'Intelligenza Artificiale. (Submitted to Neural Computation)

Siegelmann, H. (1992). *Theoretical foundations of recurrent neural networks.* Unpublished doctoral dissertation, Rutgers, New Brunswick Rutgers, The State of New Jersey.

Siegelmann, H. T., & Sontag, E. D. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, *4*(6), 77–80.

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.

Steijvers, M., & Grunwald, P. (1996). A recurrent network that performs a contextsensitive prediction task. In *Proceedings of the 18th Annual Conference of the Cognitive Science Society.* Erlbaum.

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

Sun, G. Z., Giles, C. L., Chen, H. H., & Lee, Y. C. (1993). *The neural network pushdown automaton: Model, stack and learning simulations* (Technical Report No. CS-TR-3118). University of Maryland, College Park.

Tonkes, B., & Wiles, J. (1997). Learning a context-free task with a recurrent neural network: An analysis of stability. In *Proceedings of the Fourth Biennial Conference of the Australasian Cognitive Science Society.*

Townley, S., Ilchmann, A., Weiss, M. G., McClements, W., Ruiz, A. C., Owens, D., & Praetzel-Wolters, D. (1999). *Existence and learning of oscillations in recurrent neural networks* (Tech. Rep. No. AGTM 202). Kaiserslautern, Germany: Universitaet Kaiserslautern, Fachbereich Mathematik.

Tsoi, A. C., & Back, A. D. (1994). Locally recurrent globally feedforward networks: A critical review of architectures. *IEEE Transactions on Neural Networks*, *5*(2), 229–239.

Tsung, F. S., & 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 TAB Neural Network Committee.

Tsung, F.-S., & Cottrell, G. W. (1995). Phase-space learning. In *Advances in Neural Information Processing Systems* (Vol. 7, pp. 481–488). The MIT Press.

Vesanto, J. (1997). Using the SOM and local models in time-series prediction. In *Proceedings of WSOM'97, Workshop on Self-Organizing Maps, Espoo, Finland, June 4–6* (pp. 209–214). Espoo, Finland: Helsinki University of Technology, Neural Networks Research Centre.

Vijay-Shanker, K. (1992). Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics*, *18*(4), 481–517.

Waibel, A. (1989). Modular construction of time-delay neural networksfor speech recognition [Letter]. *Neural Computation*, *1*(1), 39–46.

Wan, E. A. (1994). Time series prediction by using a connectionist network with internal time delays. In W. A. S. & G. N. A. (Eds.), *Time Series Prediction: Forecasting the Future and Understanding the Past* (pp. 195–217). Addison-Wesley.

Weigend, A., & Gershenfeld, N. (1993"). *Time series prediction: Forecasting the future and understanding the past.* Addison-Wesley.

Weigend, A. S., & Nix, D. A. (1994). Predictions with confidence intervals (local error bars). In *Proceedings of the International Conference on Neural Information Processing (ICONIP'94)* (pp. 847–852). Seoul, Korea.

Weiss, M. G. (1999). *Learning oscillations using adaptive control* (Tech. Rep. No. AGTM 178). Kaiserslautern, Germany: Universitaet Kaiserslautern, Fachbereich Mathematik.

Werbos, P. J. (1988). Generalisation of backpropagation with application to a recurrent gas market model. *Neural Networks*, *1*, 339–356.

Wiles, J., & Elman, J. (1995). Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *In Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society* (pp. pages 482 – 487). Cambridge, MA: MIT Press.

Williams, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories [Letter]. *Neural Computation*, *2*(4), 490–501.

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

Williams, R. J., & Zipser, D. (1992). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin & D. E. Rumelhart (Eds.), *Back-propagation: Theory, Architectures and Applications* (pp. 433–486). Hillsdale, NJ: Erlbaum.

Yao, X., & Liu, Y. (1997). A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, *8*(3), 694–713.

Zelle, J., & Mooney, R. (1993). Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the 11th national conference on artificial intelligence, aaai* (pp. 817–822). MIT Press.

Zeng, Z., Goodman, R., & Smyth, P. (1994"). Discrete recurrent neural networks for grammatical inference. *IEEE Transactions on Neural Networks*, *5*(2).

Personal Record

Lugano, January 30, 2001

Name	Felix Alexander Gers	
Date of birth	15.11.1970	
Place of birth	Freiburg/Br. (Germany)	
Nationality	German	
Marital status	single	
Parents	Dietmar Gers Erika Gers born Marschner	
Education	1976 - 1980	Primary school
	1980 - 1982	Orientation school
	1982 - 1989	High school (grammar-school) Bismarckschule Hannover qualification for admission to a university (Abitur)
	1989 - 1995	Study of physics at the University of Hannover
	1991	Intermediate examination
	1995	Master degree (Diplom) in physics at the University of Hannover
Work	1996 - 1997	Advanced Telecommunication Research Center (ATR, Kyoto, Japan), Human Information Processing Laboratories, Evolutionary Systems Department
	1997	Laser Zentrum Hanover (LZH), Germany, Optical Measurement Techniques Group
	1997 -	Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA, Lugano, Switzerland), Neural Network Group

Publications

Gers, F. A., Eck, D., & Schmidhuber, J. Applying LSTM to time series predictable through time-window approaches. In *Neural Nets, WIRN Vietri-99, Proceedings 11th Workshop on Neural Nets.*

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

Cummins, F., Gers, F. A., & Schmidhuber, J. (1999). *Automatic discrimination among languages based on prosody alone* (Tech. Rep. No. IDSIA-03-99). Lugano, CH: IDSIA.

De Garis, H., Gers, F. A., Korkin, M., Agah, A., & Nawa, N. E. (1998). Building an artificial brain using an FPGA based 'CAM-brain machine'. *Artificial Life and Robotics Journal*, *2*, 56-61.

Gers, F. A., & Czarske, J. W. (1995). Untersuchungen zur verteilten temperatur-sensorik mit stimulierter brillouin-streuung. In *Laser'95 Conference Proceedings C P22.*

Gers, F. A., & De Garis, H. (1996a). Porting a cellular automata based artificial brain to MIT's cellular automata machine "CAM-8". In *Int. Conf. on Simulated Evolution and Learnin (SEAL) S7-3, Taejon, Korea.*

Gers, F. A., & De Garis, H. (1996b). CAM-brain : A new model for ATR's cellular automata based artificial brain project. In *Int. Conf. on Evolvable Systems Conference Proceedings (ICES) S7-5, Tsukuba, Japan.*

Gers, F. A., & De Garis, H. (1997). Codi-1bit : A simplified cellular automata based neuron model. In *Artificial Evolution Conference (AE), Nimes, France.*

Gers, F. A., De Garis, H., & Korkin, M. (1997a). Evolution of neural sructures based on cellular automata. In C. J. Lakhmi (Ed.), *Soft computing techniques in knowlage-based intelligent engineering systems* (p. 259-278). Heidelberg New York: Physica-Verlag.

Gers, F. A., De Garis, H., & Korkin, M. (1997b). A simplified cellular automata based neuron model. In J. Hao, E. Lutton, E. Ronald, M. Schoennauer, & D. Snyers (Eds.), *Artificial Evolution* (p. 315-334). Springer Verlag.

Gers, F. A., De Garis, H., & Korkin, M. (1998). Codi-1bit : A cellular automata based neural net model simple enough to be implemented in evolvable hardware. In *Int.Symposium on Artificial Life and Robotics (AROB), Beppu, Oita, Japan.*

Gers, F. A., Eck, D., & Schmidhuber, J. (2000). *Applying LSTM to time series predictable through time-window approaches* (Tech. Rep. No. IDSIA-22-00). Manno, CH: IDSIA.

Gers, F. A., Eck, D., & Schmidhuber, J. (2001). Applying LSTM to time series predictable through time-window approaches. In *Proc. ICANN 2001, Int. Conf. on Artificial Neural Networks.* Vienna, Austria: IEE, London. (submitted)

Gers, F. A., & Schmidhuber, J. Long short-term memory learns context free and context sensitive languages. In *ICANNGA 2001 Conference.* (accepted)

Gers, F. A., & Schmidhuber, J. (2000a). LSTM learns context free languages. In *Snowbird 2000 Conference.*

Gers, F. A., & Schmidhuber, J. (2000b). *Long short-term memory learns context free languages and context sensitive languages* (Tech. Rep. No. IDSIA-03-00). Manno, CH: IDSIA.

Gers, F. A., & Schmidhuber, J. (2000c). Neural processing of complex continual input streams. In *Proc. IJCNN'2000, Int. Joint Conf. on Neural Networks.* Como, Italy.

Gers, F. A., & Schmidhuber, J. (2000d). *Neural processing of complex continual input streams* (Tech. Rep. No. IDSIA-02-00). Manno, CH: IDSIA.

Gers, F. A., & Schmidhuber, J. (2000e). Recurrent nets that time and count. In *Proc. IJCNN'2000, Int. Joint Conf. on Neural Networks.* Como, Italy.

Gers, F. A., & Schmidhuber, J. (2000f). *Recurrent nets that time and count* (Tech. Rep. No. IDSIA-01-00). Manno, CH: IDSIA.

Gers, F. A., & Schmidhuber, J. (2001). Long short-term memory learns simple context free and context sensitive languages. *IEEE Transactions on Neural Networks.* (accepted)

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999a). Continual prediction using LSTM with forget gates. In M. Marinaro & R. Tagliaferri (Eds.), *Neural Nets, WIRN Vietri-99, Proceedings 11th Workshop on Neural Nets* (p. 133-138). Vietri sul Mare, Italy: Springer Verlag, Berlin.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999b). Learning to forget: Continual prediction with LSTM. In *Proc. ICANN'99, Int. Conf. on Artificial Neural Networks* (Vol. 2, p. 850-855). Edinburgh, Scotland: IEE, London.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999c). *Learning to forget: Continual prediction with LSTM* (Tech. Rep. No. IDSIA-01-99). Lugano, CH: IDSIA.

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

Gers, F. A., Schmidhuber, J., & Schraudolph, N. Learning precise timing with LSTM recurrent networks. (submitted to Neural Computation)

Hough, M., De Garis, H., Korkin, M., Gers, F. A., & Nawa, N. E. (1999). Spiker : Analog waveform to digital spiketrain conversion in atr's artificial brain "cam-brain" project. In *Int. Conf. on Robotics and Artificial Life, Beppu, Japan.*

Korkin, M., De Garis, H., Gers, F., & Hemmi, H. (1997). 'CBM (CAM-brain machine) : A hardware tool which evolves a neural net module in a fraction of a second and runs a million neuron artificial brain in real time. In *Genetic Programming Conference, Stanford, USA.*

Nawa, N. E., De Garis, H., Gers, F. A., & Korkin, M. (1998). 'ATR's CAM-brain machine (CBM) simulation results and representation issues. In *Genetic Programming Conference.*

Acknowledgments

I am grateful to everybody who helped me to start, do and finish this thesis.

Special thanks go to my parents who always supported me in everything I wanted to do.

This thesis was only possible, because Juergen Schmidhuber set up the LSTM project at IDSIA including the position that I took in the last years. During all my time at IDSIA Juergen always left me the freedom to follow my own ideas. He was excellent as scientific reference point and as critic to test my ideas against. I greatly appreciated working with him.

I want to thank Wulfram Gerstner for accepting the supervision over the theses. His critical feedback was always very helpful for my work.

I always enjoyed exchanging ideas with Doug and Fred, who worked with me on the LSTM project.

My deep thanks go to Mara for all the things she did for me during my time in Lugano. Rafal accompanied me, working on his thesis, from the day of my interview until now, and I hope we can also celebrate the "the days after" (for both of us) together. Nic was always there for any scientific discussion down into painful details. Everything would have been much more difficult without Ivo driving me through the last part of the thesis while I could not walk. Marco "Zaffa" helped to transform my Italian into Italian.

Thanks to everybody at IDSIA and Old-IDSIA for creating a great atmosphere for working with lots of fun.

The persons I mentioned, but also many others outside and around IDSIA did much more for me when I want to write here. They know.