Journal of Machine Learning Research 3 (2002) 115-143 Submitted 5/01; Revised 3/02; Published 8/02

Learning Precise Timing with LSTM Recurrent Networks

Felix A. Gers FELIX@IDSIA.CH
IDSIA
Galleria 2
6928 Manno, Switzerland
www.idsia.ch

Nicol N. Schraudolph SCHRAUDO@INF.ETHZ.CH
Inst. of Computational Science
ETH Zentrum
8092 Zürich, Switzerland
www.icos.ethz.ch

Jürgen Schmidhuber JUERGEN@IDSIA.CH
IDSIA
Galleria 2
6928 Manno, Switzerland
www.idsia.ch

Editor: Michael I. Jordan

Abstract

The temporal distance between events conveys information essential for numerous sequential tasks such as motor control and rhythm detection. While Hidden Markov Models tend to ignore this information, recurrent neural networks (RNNs) can in principle learn to make use of it. We focus on Long Short-Term Memory (LSTM) because it has been shown to outperform other RNNs on tasks involving long time lags. We find that LSTM augmented by "peephole connections" from its internal cells to its multiplicative gates can learn the fine distinction between sequences of spikes spaced either 50 or 49 time steps apart without the help of any short training exemplars. Without external resets or teacher forcing, our LSTM variant also learns to generate stable streams of precisely timed spikes and other highly nonlinear periodic patterns. This makes LSTM a promising approach for tasks that require the accurate measurement or generation of time intervals.

Keywords: Recurrent Neural Networks, Long Short-Term Memory, Timing.

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 paper, 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.

Recurrent neural networks (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 and 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 and Frasconi, 1995) may 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.

We have previously introduced a novel type of RNN called Long Short-Term Memory (LSTM—Hochreiter and Schmidhuber, 1997) that works better than traditional RNNs on tasks involving long time lags. Its architecture permits LSTM to bridge huge time lags between relevant input events (1000 steps and more), while traditional RNNs with more costly update algorithms such as BPTT (Williams and Peng, 1990), RTRL (Robinson and Fallside, 1987, Williams and Zipser, 1992), or combinations thereof (Schmidhuber, 1992, Williams and Zipser, 1992), already fail to learn in the presence of 10-step time lags (Hochreiter, 1991, Bengio et al., 1994, Hochreiter and Schmidhuber, 1997, Gers et al., 2000, Hochreiter et al., 2001).

For instance, some of our previous tasks required an LSTM network to act upon events that occurred 50 discrete time steps ago, independently of what happened over the intervening 49 steps. Right before the critical moment, however, there was a helpful "marker" input informing the network that its next action would be crucial. Thus the network did not 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.

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 paper 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. Clearly, such tasks cannot generally be solved by common time-window based approaches, because their generalization ability is limited by the time window size. Unfortunately, this means that we cannot use standard benchmarks for measuring and timing because there aren't any—to our knowledge no other network has yet learned to generalize from time lags of size 15 to time lags of size 45, etc. Hence we are forced to create a new set of comparative tasks.

Outline. Section 2 gives an overview of "traditional" LSTM, as used in our previous publications. Section 3 identifies a weakness in its connection scheme, introduces "peephole connections" as a remedy, and describes the modifications to the LSTM learning algorithm they necessitate. Section 4 compares the performance of peephole LSTM to traditional

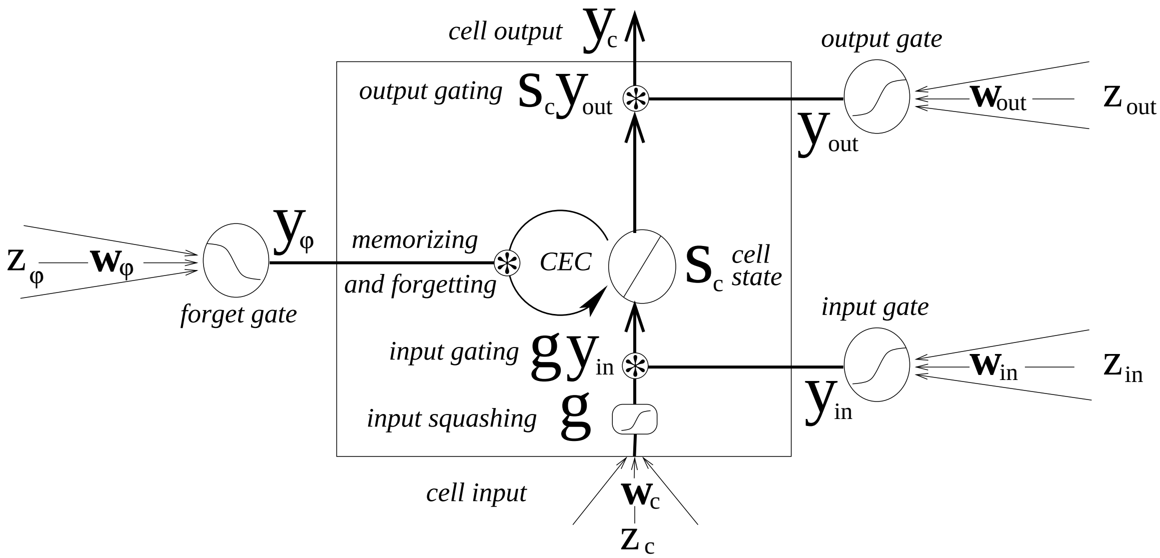

Figure 1: LSTM memory block with one cell (rectangle). The so-called CEC maintains the cell state s_c, which may be reset by the forget gate. Input and output gate control read and write access to the CEC; g squashes the cell input. See text for details.

LSTM on timings tasks of the kind described above, and Section 5 provides some further discussion of our approach.

2. Traditional LSTM

We are building on LSTM with forget gates (Gers et al., 2000), simply called "LSTM" in what follows. The basic unit of an LSTM network is the *memory block* containing one or more *memory cells* and three adaptive, multiplicative gating units shared by all cells in the block (Figure 1). Each memory cell has at its core a recurrently self-connected linear unit we call the "Constant Error Carousel" (CEC). By recirculating activation and error signals indefinitely, the CEC provides short-term memory storage for extended time periods. The *input*, *forget*, and *output gate* can be trained to learn, respectively, what information to store in the memory, how long to store it, and when to read it out. Combining memory cells into blocks allows them to share the same gates (provided the task permits this), thus reducing the number of adaptive parameters.

Throughout this paper 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. The output y_c of memory cell c is calculated based on the current cell state s_c and four sources of input: z_c is the input to the cell itself, while z_{in}, z_φ and z_{out} feed into the input, forget, and output gate, respectively. LSTM operates in discrete time steps $t = 0, 1, 2, \ldots$, each involving the update of all units' activation (forward pass) followed by the computation of error signals for all weights (backward pass).

2.1 Forward Pass

Input. During each forward pass we first calculate the net cell input

$$z_{c_j^v}(t) \;=\; \sum_m w_{c_j^v m}\; y_m(t-1)\,, \tag{1}$$

then apply the (optional) *input squashing* function g to it. The result is multiplied by the activation of the memory block's input gate, calculated by applying a logistic sigmoid squashing function f_{in} with range $[0,1]$ to the gate's net input z_{in}:

$$y_{\text{in}_j}(t) \;=\; f_{\text{in}_j}(z_{\text{in}_j}(t))\,, \qquad z_{\text{in}_j}(t) \;=\; \sum_m w_{\text{in}_j m}\; y_m(t-1)\,. \tag{2}$$

The activation y_{in} of the input gate multiplies the input to all cells in the memory block, and thus determines which activity patterns are stored (added) into it. During training, the input gate learns to open ($y_{\text{in}} \approx 1$) so as to store relevant inputs in the memory block, respectively close ($y_{\text{in}} \approx 0$) so as to shield it from irrelevant ones.

Cell State. At $t = 0$, the activation (or *state*) s_c of a memory cell c is initialized to zero; subsequently the CEC accumulates a sum, discounted by the forget gate, over its input. Specifically, we first calculate the memory block's forget gate activation

$$y_{\varphi_j}(t) \;=\; f_{\varphi_j}(z_{\varphi_j}(t))\,, \qquad z_{\varphi_j}(t) \;=\; \sum_m w_{\varphi_j m}\; y_m(t-1)\,, \tag{3}$$

where f_φ is a logistic sigmoid function with range $[0,1]$. The new cell state is then obtained by adding the squashed, gated cell input to the previous state multiplied by the forget gate activation:

$$s_{c_j^v}(t) \;=\; y_{\varphi_j}(t)\; s_{c_j^v}(t-1) \;+\; y_{\text{in}_j}(t)\; g(z_{c_j^v}(t))\,, \qquad s_{c_j^v}(0) \;=\; 0\,. \tag{4}$$

Thus activity circulates in the CEC as long as the forget gate remains open ($y_\varphi \approx 1$). Just as the input gate learns what to store in the memory block, the forget gate learns for how long to retain the information, and—once it is outdated—to erase it by resetting the cell state to zero. This prevents the cell state from growing to infinity, and enables the memory block to store fresh data without undue interference from prior operations (Gers et al., 2000).

Output. The cell output y_c is calculated by multiplying the cell state s_c by the activation y_{out} of the memory block's output gate:

$$y_{c_j^v}(t) \;=\; y_{\text{out}_j}(t)\; s_{c_j^v}(t)\,. \tag{5}$$

Here we introduce a minor simplification unrelated to the central idea of this paper. Traditional LSTM memory cells incorporate an input squashing function g *and* an output squashing function (called h in earlier LSTM publications); we remove the latter from equation 5 for lack of empirical evidence that it is needed.

By multiplying the output from the CEC, the output gate controls read access to the memory block. Its activation is calculated by applying a logistic sigmoid squashing function f_{out} with range $[0, 1]$ to its net input:

$$y_{\mathrm{out}_j}(t) \;=\; f_{\mathrm{out}_j}(z_{\mathrm{out}_j}(t))\,, \qquad z_{\mathrm{out}_j}(t) \;=\; \sum_m w_{\mathrm{out}_j m}\; y_m(t-1)\,. \tag{6}$$

Finally, assuming a layered network topology with a standard input layer, a hidden layer consisting of memory blocks, and a standard output layer, the activation of the output units k is calculated as:

$$y_k(t) \;=\; f_k(z_k(t))\,, \qquad z_k(t) \;=\; \sum_m w_{km}\; y_m(t)\,, \tag{7}$$

where m ranges over all units feeding the output units, and f_k is the output squashing function. This concludes the forward pass of a traditional LSTM network.

2.2 Gradient-Based Backward Pass

LSTM's backward pass is an efficient fusion of error back-propagation (BP) for output units and output gates, and a customized, truncated version of real-time recurrent learning (RTRL—e.g., Robinson and Fallside, 1987, Williams and Zipser, 1992) for weights to cell input, input gates, and forget gates. Here we present the equations necessary to implement the LSTM backward pass; see Gers et al. (2000) for a full derivation of the algorithm.

Output units and gates. Following previous notation (Gers et al., 2000), 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:

$$\Delta w_{km}(t) \;=\; \alpha\; \delta_k(t)\; y_m(t-1)\,, \qquad \delta_k(t) \;=\; -\frac{\partial E(t)}{\partial z_k(t)}\,. \tag{8}$$

Here we use the customary squared error objective function based on targets t_k, yielding:

$$\delta_k(t) \;=\; f'_k(z_k(t))\; e_k(t)\,, \tag{9}$$

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 source units m (as specified by the network topology) are also obtained by standard back-propagation:

$$\Delta w_{\mathrm{out}_j m}(t) \;=\; \alpha\; \delta_{\mathrm{out}_j}(t)\; y_m(t)\,, \tag{10}$$

$$\delta_{\mathrm{out}_j}(t) \;\stackrel{tr}{=}\; f'_{\mathrm{out}_j}(z_{\mathrm{out}_j}(t)) \left(\sum_{v=1}^{S_j} s_{c_j^v}(t) \sum_k w_{k c_j^v}\; \delta_k(t) \right), \tag{11}$$

where the $\stackrel{tr}{=}$ sign indicates error truncation (see below).

Error flow and truncation. An error signal arriving at a memory cell output is scaled by the output gate before it enters the memory cell's CEC, where it can flow back indefinitely without ever being changed as long as the forget gate's activation remains near 1.0 (which is why LSTM can bridge arbitrary time lags between input events and target signals). When the error escapes from the memory cell through an open input gate and the input nonlinearity g, it gets scaled once more and then serves to change incoming weights before being truncated. Truncation means that errors arriving at net inputs of memory blocks and their gates do not get propagated back further in time (though they *do* serve to change the incoming weights); it makes the LSTM learning algorithm very efficient.

In conventional RNN architectures, truncating errors in this fashion would preclude the network from learning problems with time lags exceeding the truncation horizon. LSTM, by contrast, has an alternative (arguably better) mechanism for bridging long time lags—the CEC—and consequently is not so much affected by error truncation. In previous experiments Hochreiter and Schmidhuber (1997) found that error truncation did not significantly worsen the LSTM network's performance.

Truncated RTRL partials. RTRL requires the forward propagation in time of certain partial derivatives. During each step, no matter whether a target is given or not, we therefore need to update the partials $\partial s_{c_j^v} / \partial w_{lm}$ for weights to the cell ($l = c_j^v$), to the input gate ($l = in$), and to the forget gate ($l = \varphi$):

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} \, y_{\varphi_j}(t) \; + \; g'(z_{c_j^v}(t)) \; y_{\text{in}_j}(t) \; y_m(t-1) \,, \tag{12}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\text{in}_j m}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\text{in}_j m}} \, y_{\varphi_j}(t) \; + \; g(z_{c_j^v}(t)) \; f'_{\text{in}_j}(z_{\text{in}_j}(t)) \; y_m(t-1) \,, \tag{13}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} \, y_{\varphi_j}(t) \; + \; s_{c_j^v}(t-1) \; f'_{\varphi_j}(z_{\varphi_j}(t)) \; y_m(t-1) \,. \tag{14}$$

Initially these partials are set to zero: $\partial s_{c_j^v}(0) / \partial w_{lm} = 0$ for $l \in \{\text{in}, \varphi, c_j^v\}$.

RTRL weight changes. To calculate weight changes Δw_{lm} 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 12, 13, and 14:

$$\Delta w_{c_j^v m}(t) = \alpha \; e_{s_{c_j^v}}(t) \; \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \,, \tag{15}$$

$$\Delta w_{\text{in}_j m}(t) = \alpha \sum_{v=1}^{S_j} \; e_{s_{c_j^v}}(t) \; \frac{\partial s_{c_j^v}(t)}{\partial w_{\text{in}_j m}} \,, \tag{16}$$

$$\Delta w_{\varphi_j m}(t) = \alpha \sum_{v=1}^{S_j} \; e_{s_{c_j^v}}(t) \; \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} \,, \tag{17}$$

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

$$e_{s_{c_j^v}}(t) \stackrel{tr}{=} y_{\text{out}_j}(t) \left(\sum_k w_{k c_j^v} \; \delta_k(t) \right) . \tag{18}$$

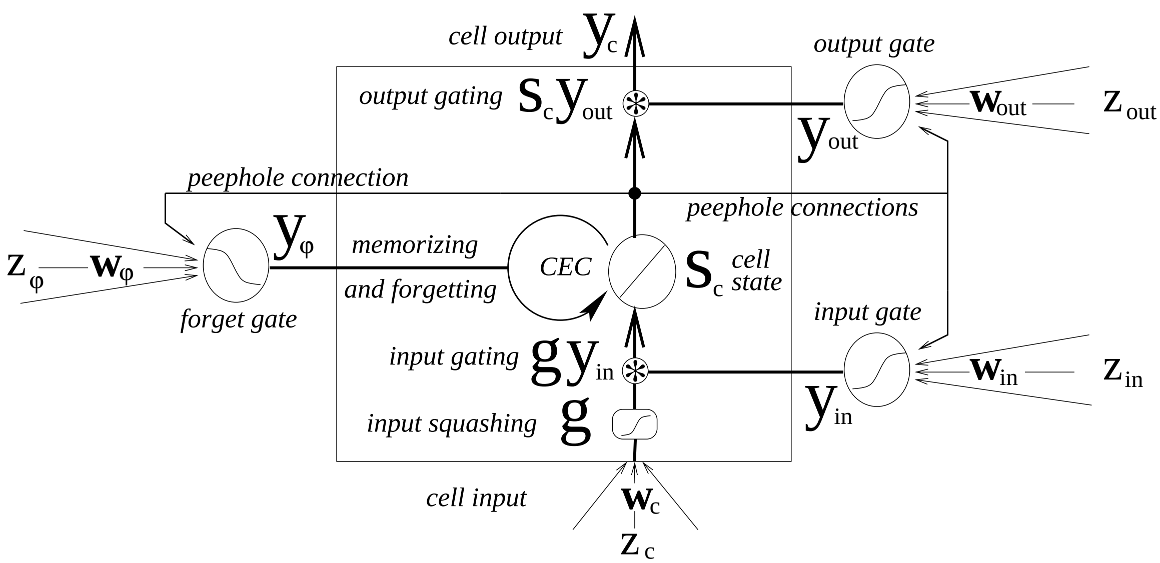

Figure 2: LSTM memory block with peephole connections from the CEC to the gates.

3. Extending LSTM with Peephole Connections

A Limitation of traditional LSTM. In traditional 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 (Figure 1). All it can observe directly is the cell output, which is close to zero as long as the output gate is closed. The same problem occurs for multiple cells in a memory block: when the output gate is closed none of the gates has access to the CECs they control. The resulting lack of essential information may harm network performance, especially in the tasks we are studying here.

Peephole connections. Our simple but effective remedy is to add weighted "peephole" connections from the CEC to the gates of the same memory block (Figure 2). The peephole connections allow all gates to inspect the current cell state even when the output gate is closed. This information can be essential for finding well-working network solutions, as we will see in the experiments below.

During learning no error signals are propagated back from gates via peephole connections to the CEC (see backward pass, Section 3.2). 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.

Update order for peephole LSTM. Each memory cell component should be updated based on the most recent activations 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 only after the cell state s_c, so that it sees via its peephole connection the current value of $s_c(t)$, already affected by input and forget gate.

Below we present the modifications and additions to the LSTM forward and backward pass, respectively, due to the presence of peephole connections. Appendix A gives the entire learning algorithm for peephole LSTM in pseudo-code.

3.1 Modified Forward Pass for Peephole LSTM

Steps 1a, 1b. The input and forget gate activation are computed as:

$$y_{\text{in}_j}(t) \;=\; f_{\text{in}_j}(z_{\text{in}_j}(t))\,, \quad z_{\text{in}_j}(t) \;=\; \sum_m w_{\text{in}_j m}\; y_m(t-1) \;+\; \sum_{v=1}^{S_j} w_{\text{in}_j c_j^v}\; s_{c_j^v}(t-1)\,; \qquad (19)$$

$$y_{\varphi_j}(t) \;=\; f_{\varphi_j}(z_{\varphi_j}(t))\,, \quad z_{\varphi_j}(t) \;=\; \sum_m w_{\varphi_j m}\; y_m(t-1) \;+\; \sum_{v=1}^{S_j} w_{\varphi_j c_j^v}\; s_{c_j^v}(t-1)\,. \qquad (20)$$

The peephole connections for input and forget gate are incorporated in Equations 19 and 20 by including the CECs of memory block j as source units. These equations replace Equations 2 and 3 of traditional LSTM.

Step 1c. The cell input and cell state s_c are calculated as before via Equations 1 and 4.

Step 2. The output gate activation y_{out} with peephole connections is computed as:

$$y_{\text{out}_j}(t) \;=\; f_{\text{out}_j}(z_{\text{out}_j}(t))\,, \quad z_{\text{out}_j}(t) \;=\; \sum_m w_{\text{out}_j m}\; y_m(t-1) \;+\; \sum_{v=1}^{S_j} w_{\text{out}_j c_j^v}\; s_{c_j^v}(t)\,. \qquad (21)$$

Equation 21 replaces Equation 6 of traditional LSTM. The output of the memory cell, and of the entire LSTM network, are then calculated as before via Equations 5 and 7, respectively.

3.2 Update Rules for Peephole Connections

The revised update scheme for memory blocks allows for treating peephole connections like regular connections, so the equations given for traditional LSTM in Section 2.2 need only be supplemented with corresponding update rules for the partial derivatives and weights associated with peephole connections. Analogous to Equations 13 and 14, we now also maintain partial derivatives for the peephole connections to input gate and forget gate:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\mathrm{in}_j c_j^{v'}}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\mathrm{in}_j c_j^{v'}}} \, y_{\varphi_j}(t) \; + \; g(z_{c_j^v}(t)) \; f'_{\mathrm{in}_j}(z_{\mathrm{in}_j}(t)) \; s_{c_j^{v'}}(t-1)\,, \tag{22}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j c_j^{v'}}} \stackrel{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j c_j^{v'}}} \, y_{\varphi_j}(t) \; + \; s_{c_j^v}(t-1) \; f'_{\varphi_j}(z_{\varphi_j}(t)) \; s_{c_j^{v'}}(t-1)\,, \tag{23}$$

with $\partial s_{c_j^v}(0)/\partial w_{\mathrm{in}_j c_j^{v'}} = \partial s_{c_j^v}(0)/\partial w_{\varphi_j c_j^{v'}} = 0$. The changes to the peephole connection weights are calculated the same way as in Equations 10, 16, and 17:

$$\Delta w_{\mathrm{out}_j c_j^v}(t) = \alpha \, \delta_{\mathrm{out}_j}(t) \; s_{c_j^v}(t)\,, \tag{24}$$

$$\Delta w_{\mathrm{in}_j c_j^{v'}}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \, \frac{\partial s_{c_j^v}(t)}{\partial w_{\mathrm{in}_j c_j^{v'}}}\,, \tag{25}$$

$$\Delta w_{\varphi_j c_j^{v'}}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \, \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j c_j^{v'}}}\,. \tag{26}$$

The storage complexity of the backward pass does not depend on the length of the input sequence. Like standard LSTM, but unlike BPTT and RTRL, LSTM with forget gates and peephole connections is local in space *and* time. The increase in complexity due to peephole connections is small: 3 weights per cell.

4. Experiments

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

Measuring spike delays (MSD). See Section 4.2. 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 4.3. 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 and Zipser, 1989, Doya and Yoshizawa, 1989, Tsung and Cottrell, 1995), GTS is highly nonlinear and involves long time lags between significant output changes, which we cannot expect to be learned by

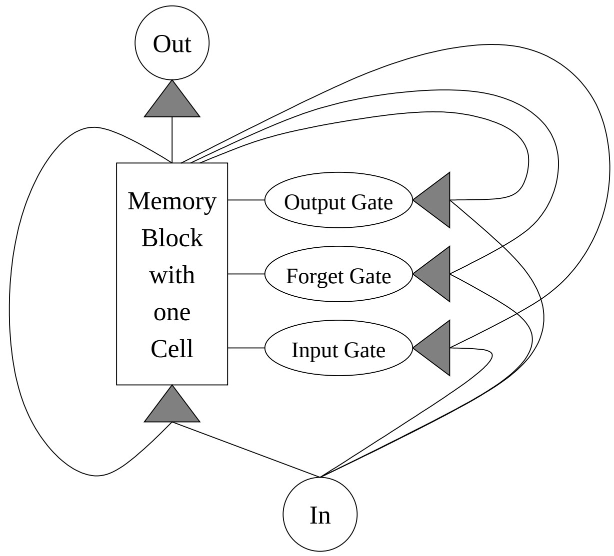

Figure 3: Three-layer LSTM topology with one input and one output unit. Recurrence is limited to the hidden layer, which consists of a single LSTM memory block with a single cell. All 9 "unit-to-unit" connections are shown, but bias and peephole connections are not.

conventional RNNs. In contrast to previous work, which did not focus on stability issues, here 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 4.4. 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 and 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 and Yoshizawa, 1989, Tsung and 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 (Hochreiter, 1991, Bengio et al., 1994, Hochreiter et al., 2001). Again we demand that the generation be stable for 1000 successive periods of the waveform.

4.1 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 3). 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 a test stream. Training and test stream are generated stochastically in the same manner. 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 et al., 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.

4.2 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 from a fixed set, randomly reset for each spike:

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

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 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

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 1: Results comparing traditional 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.

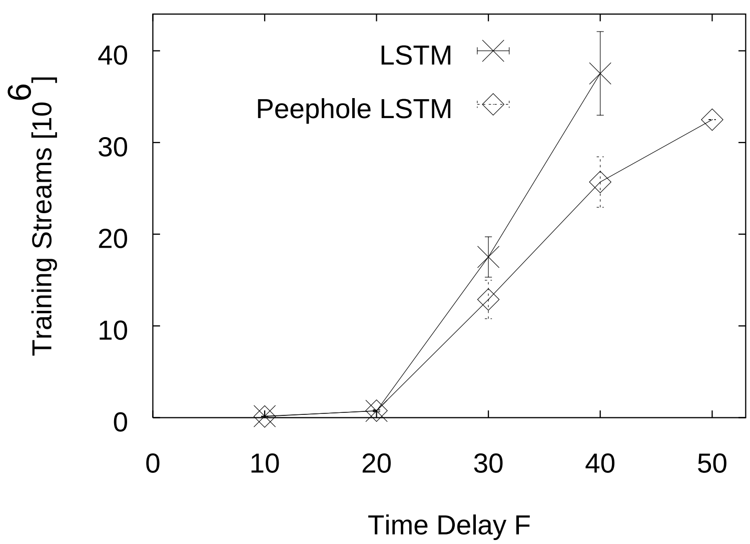

Figure 4: Average number of training streams required for the NMSD task with $I(n) \in \{0,1\}$, plotted against the minimum spike interval F.

4). A qualitative explanation is that longer intervals necessitate finer tuning of the weights, which requires more training. Peephole LSTM outperforms LSTM for some sets, though peephole connections are not mandatory for the task. They correlate the opening of the output gate to high cell states. Otherwise the output gate has to learn to be open all the time, using its bias input, which may take longer, because the cells states might have higher activation values than the bias (activation one). 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 1).

$I(n) \in$	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 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 $I(n)$.

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 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 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 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.

We also observe that the results for $I(n) \in \{0,1\}$ are better than those obtained with a sigmoid function (compare with Table 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 et al., 1998).

MSD Analysis. LSTM learned to measure time in two ways. The first is to slightly increase the cell state s_c at each time step, so that the elapsed time can be told by the value of s_c. This kind of solution is shown on the left-hand side of Figure 5. (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). 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).

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. In that case it becomes mandatory to realize a solution where the output gate only opens when a target is requested, which cannot be done without peephole connections. Figure 6 compares the behavior of LSTM, where the output gate is open over long periods, to peephole LSTM, where the output gate opens only when a target is provided. In some cases the "cleaner" solutions with peephole connections took longer to be learned (see Tables 1 and 2), because they require more complex behavior. Overall there is no clear statistical advantage either for or against using peephole connections on the MSD task.

4.3 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 7) suggest that the required training time increases dramatically with F, as with the NMSD task (Section 4.2). The network output during a successful test run for the GTS task with $F=10$ is shown on the top left of Figure 8. Peephole LSTM also solves the task for $F=10$ and $I(n) \in \{0,1\}$ or $\{0,1,2\}$, as shown in Figure 7 (left).

GTS Analysis. Figure 8 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 8). 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 8 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

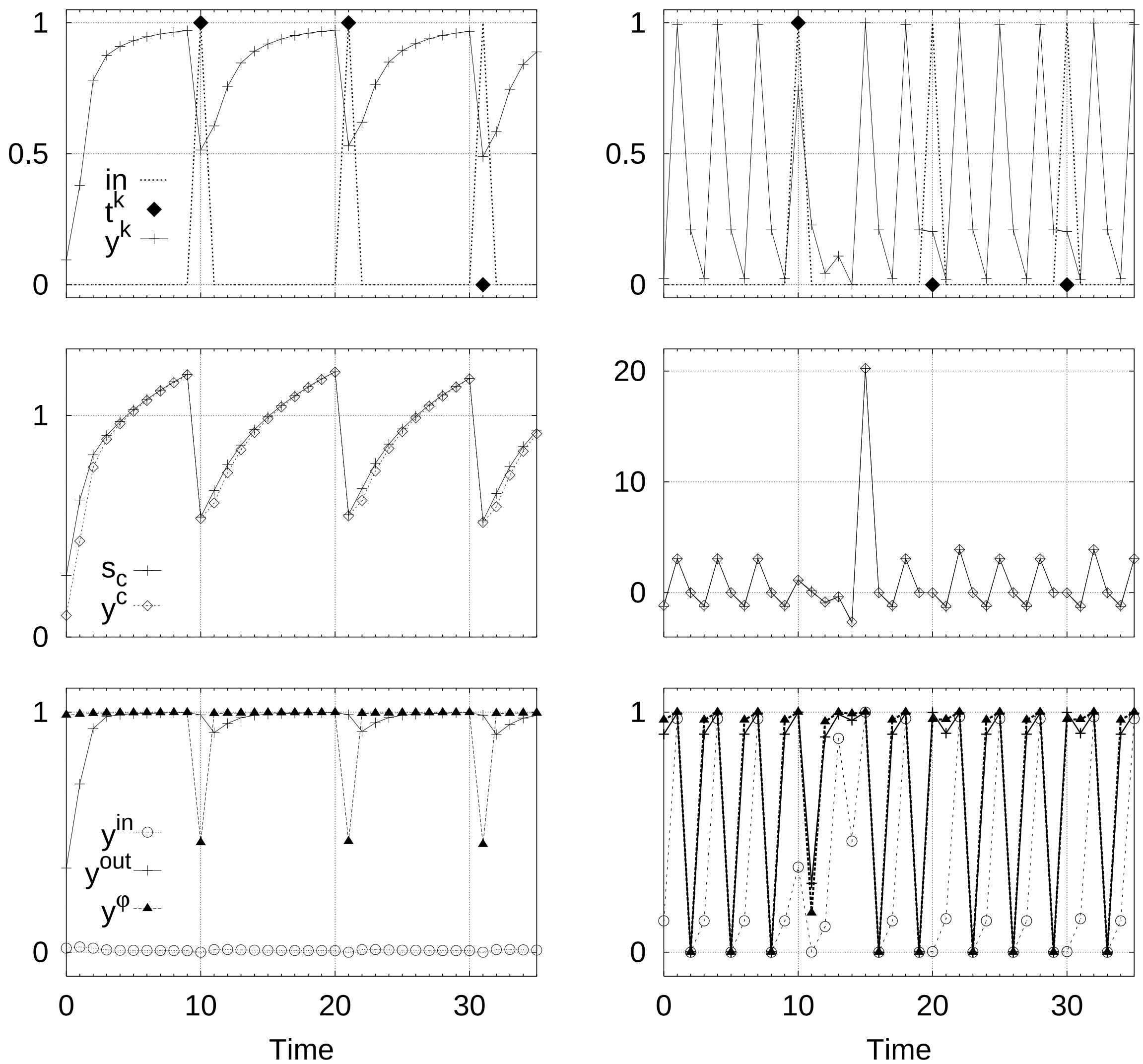

Figure 5: **Two ways to time.** Test run with trained LSTM networks for the MSD task with $F = 10$ and $I(n) \in \{0, 1\}$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_{φ}, and output gate y_{out}.

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 $I(n) \in \{0, 1\}$

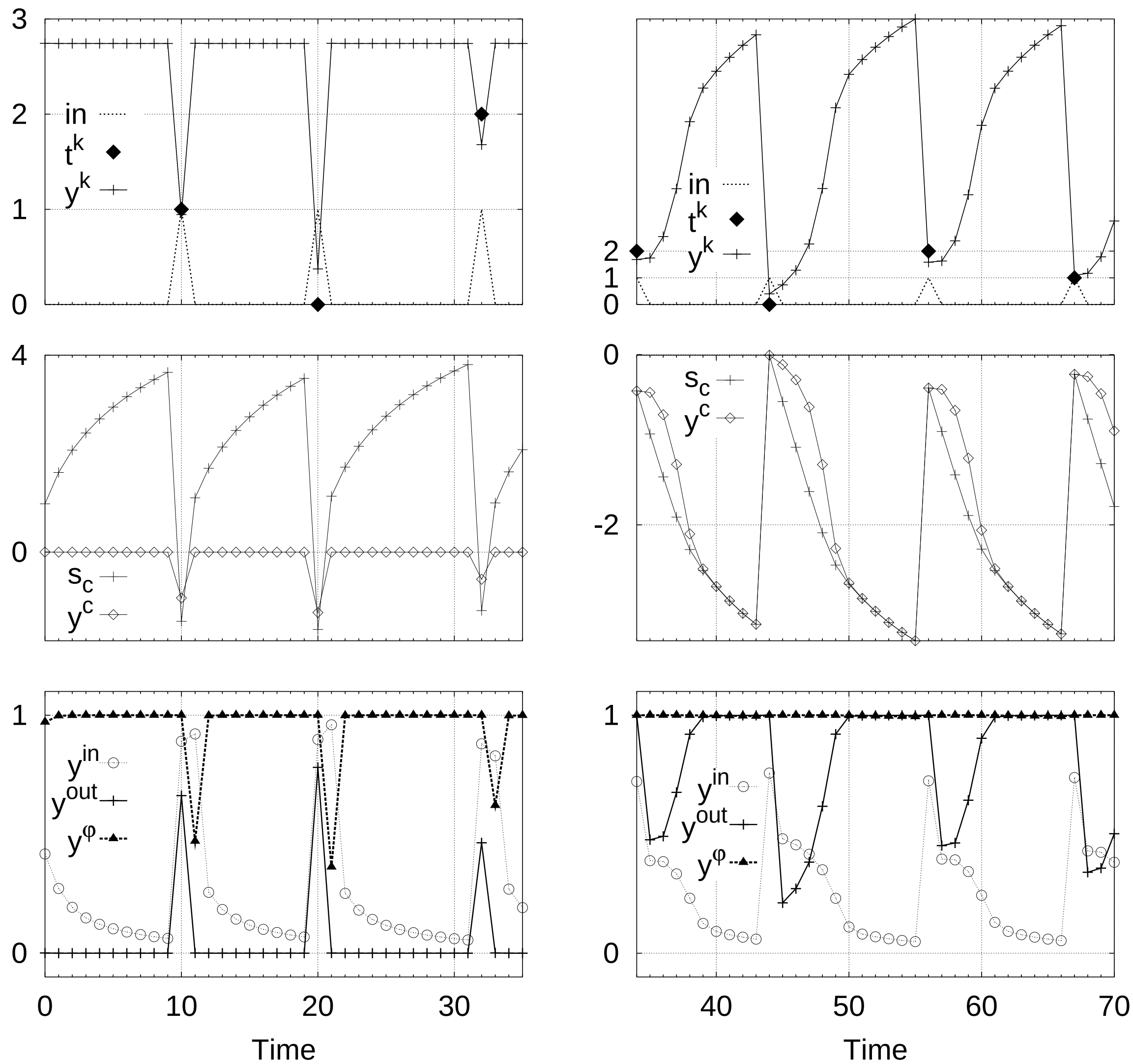

Figure 6: Behavior of peephole LSTM (left) versus LSTM (right) for the MSD task with $F=10$ and $I(n) \in \{0, 1, 2\}$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_{φ}, and output gate y_{out}.

or $I(n) \in \{0, 1, 2\}$, the network essentially translates the input into a scaling factor for the growth of s_c (Figure 9).

4.4 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.

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

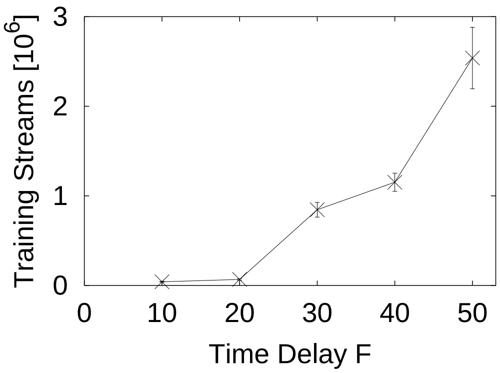

Figure 7: Results for the GTS task. Table (left) shows the minimum spike interval F, set of delays $I(n)$, percentage of perfect solutions found, and mean and standard derivation of the number of training streams required. Graph (right) plots the number of training streams against the minimum spike interval F, for $I(n) \in \{0\}$.

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

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

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

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

PFG Results. Our experimental results for the PFG task are summarized in Table 3. Peephole LSTM found perfect, stable solutions for all target functions (Figure 10). 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.

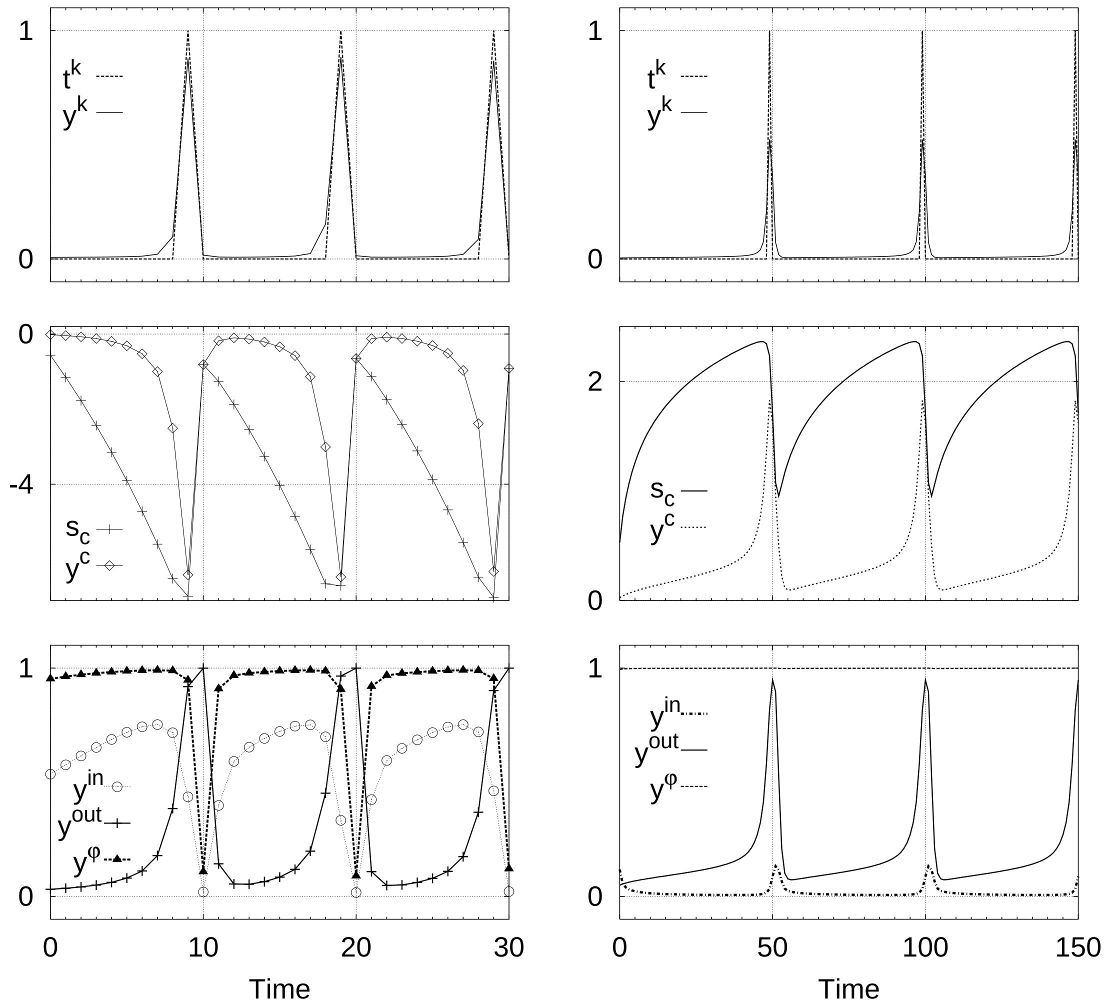

Figure 8: Test run of a trained peephole LSTM network for the GTS task with $I(n) \in \{0\}$, and a minimum spike interval of $F = 10$ (left) vs. $F = 50$ (right). Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

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.

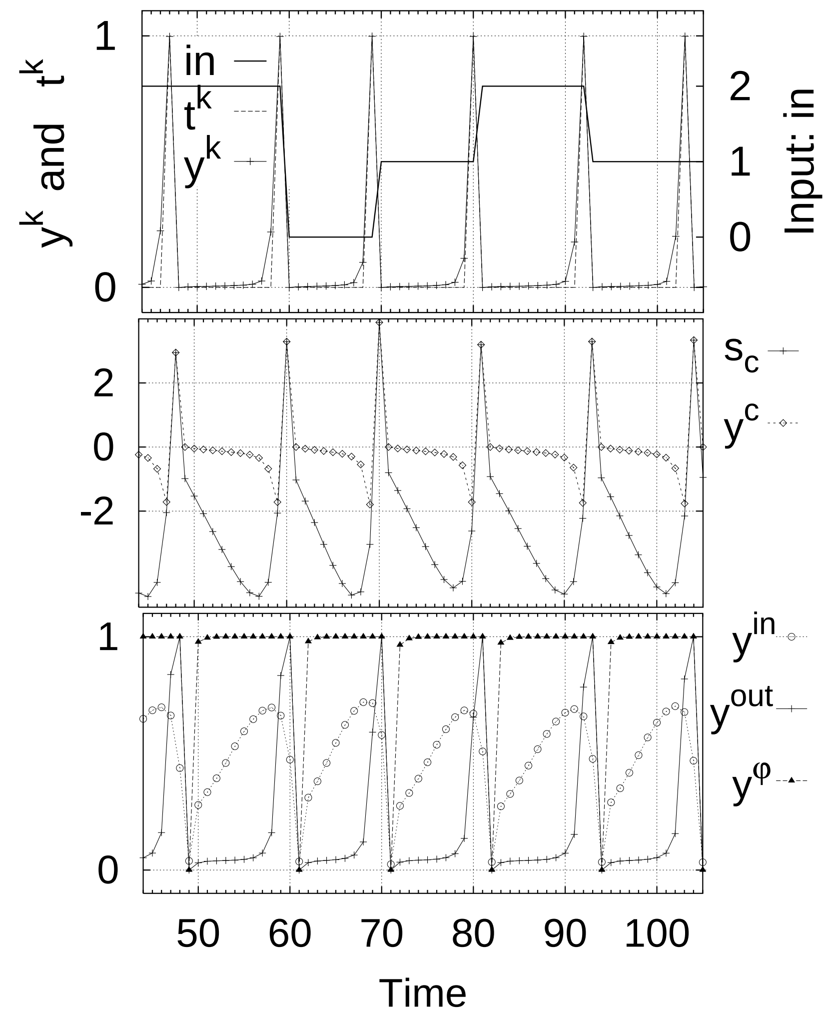

Figure 9: Test run of a trained peephole LSTM network for the GTS task with $F = 10$ and $I(n) \in \{0, 1, 2\}$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

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 11 for $f_{\cos}$ with $F = 10$. This task can be solved with or

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 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.

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 3). The peephole weights of solutions are typically of the same magnitude as the weights of 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 12 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 4.2. 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 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.

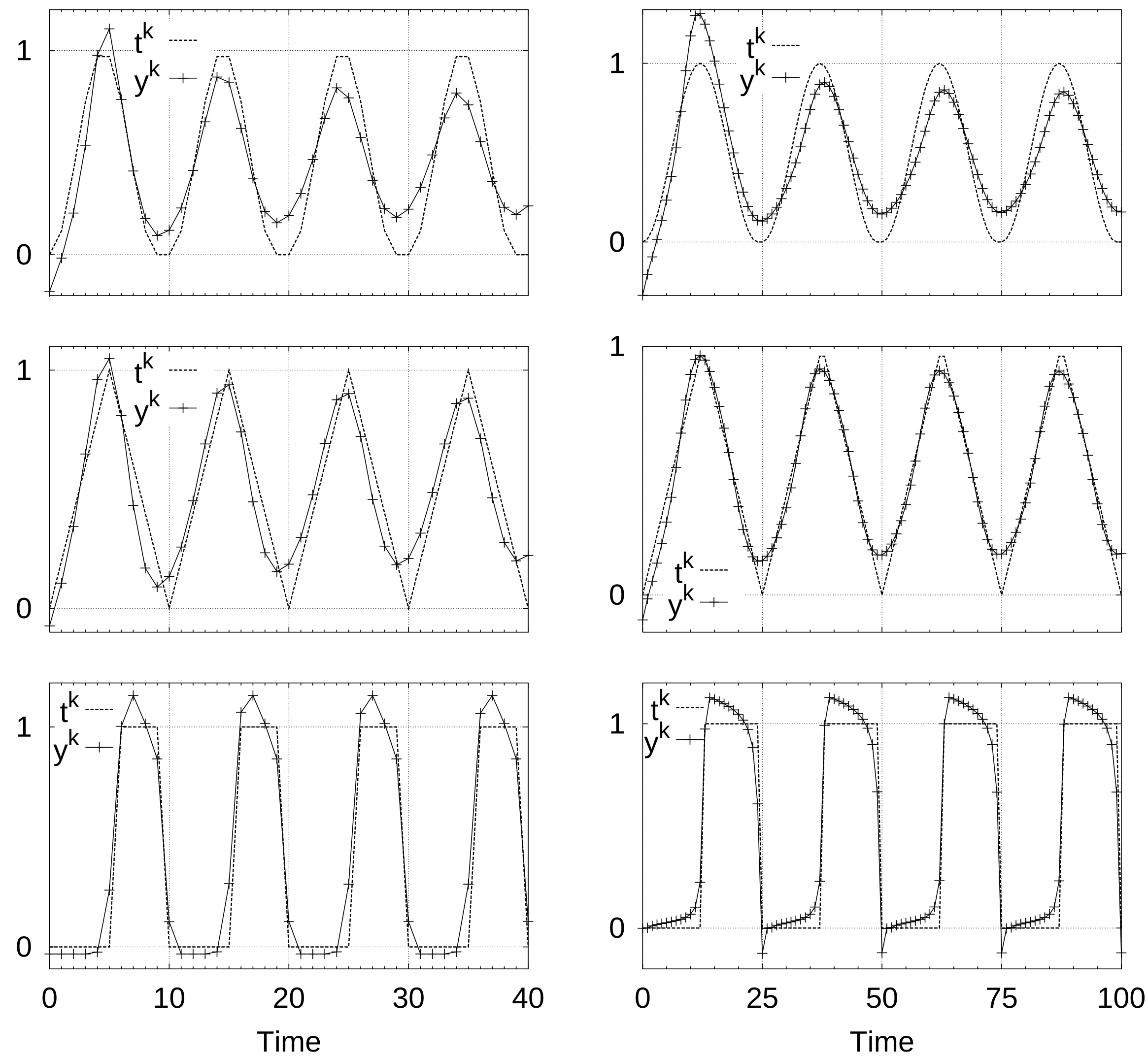

Figure 10: Target values t_k and network output y_k during test runs of trained peephole LSTM networks on the PFG task for the periodic functions $f_{\cos}$ (top), f_{tri} (middle), and f_{rect} (bottom), with periods $F = 10$ (left) and $F = 25$ (right).

5. Discussion

In this section we will discuss limitations and possible improvements of LSTM and give final conclusions.

5.1 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

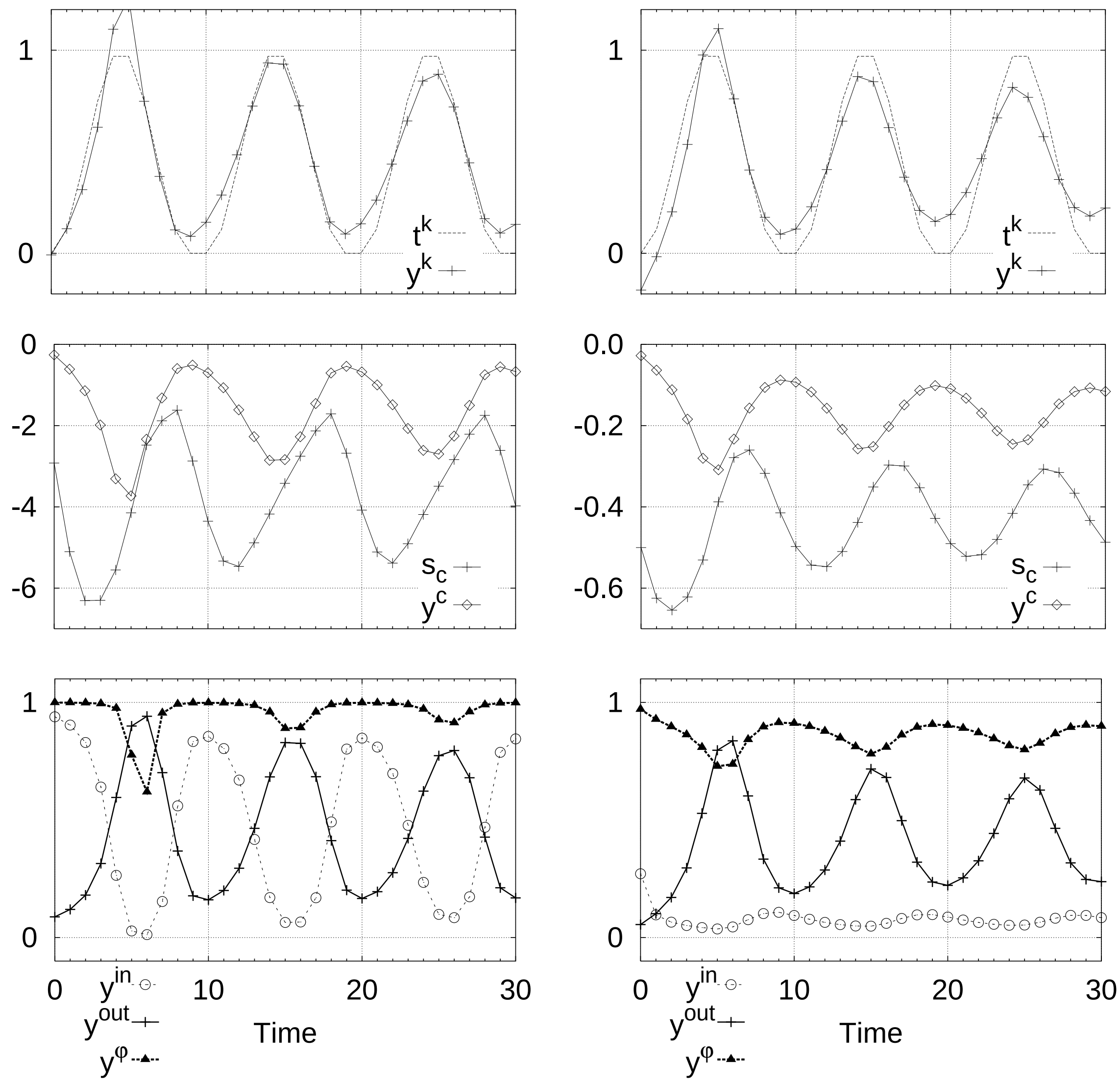

Figure 11: Test runs of a trained LSTM network with (right) vs. without (left) peephole connections on the $f_{\cos}$ PFG task with $F = 10$. Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

later periods in the stream. Figure 13 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), Bulsari and Saxén (1995), instead of arbitrarily initializing it to zero.

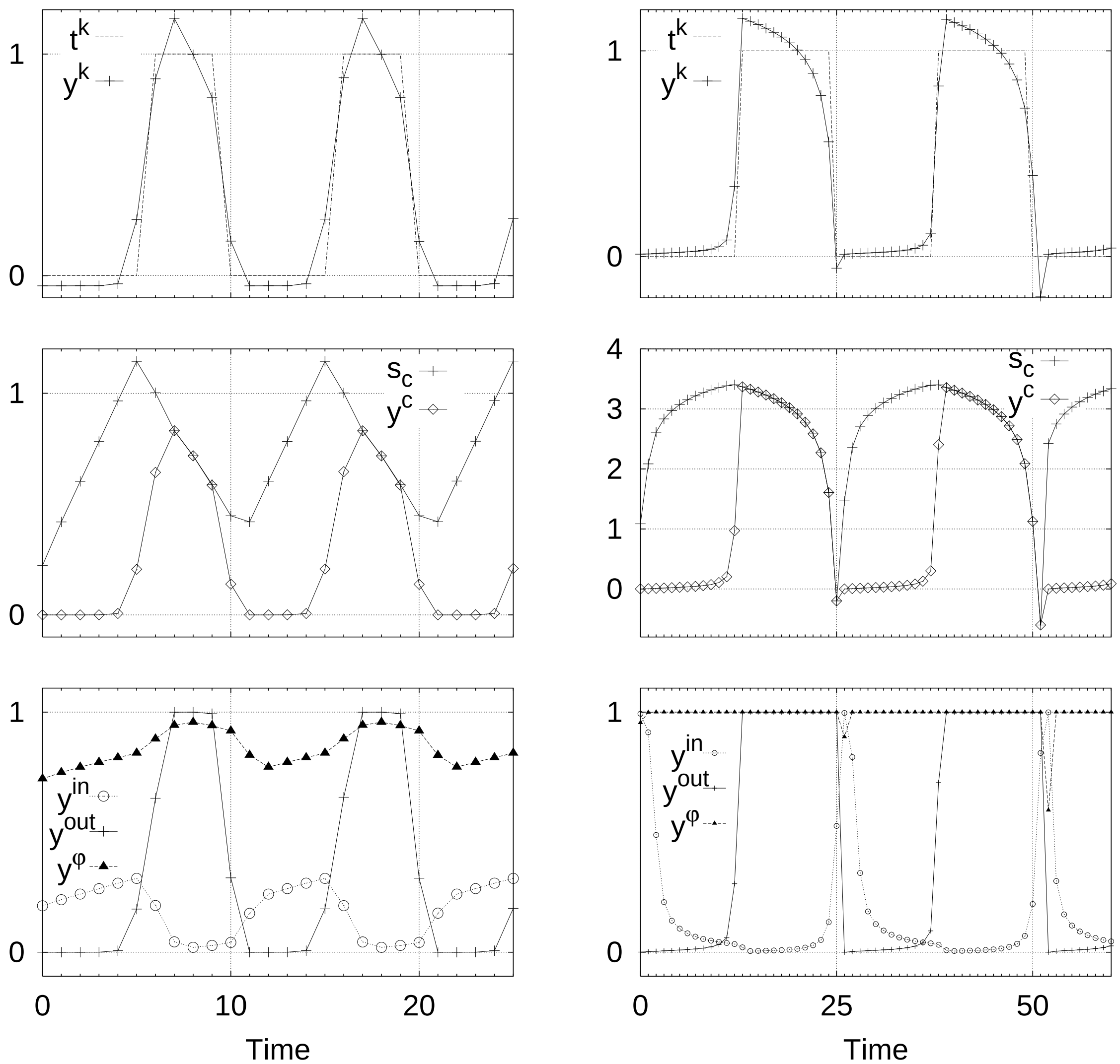

Figure 12: Test runs of trained peephole LSTM networks on the f_{rect} PFG task with $F = 10$ (left) and $F = 25$ (right). Top: target values t_k and network output y_k; middle: cell state s_c and cell output y_c; bottom: activation of the input gate y_{in}, forget gate y_φ, and output gate y_{out}.

5.2 Limitations of LSTM

LSTM excels on tasks in which a limited amount of data (which need *not* be specially marked though) must be remembered for a long time. Where additional processing of short-term information is necessary LSTM performs at least as well as other RNN algorithms (Hochreiter and Schmidhuber, 1997, Gers et al., 2000). Here we briefly point out the limitations of LSTM, and RNNs in general.

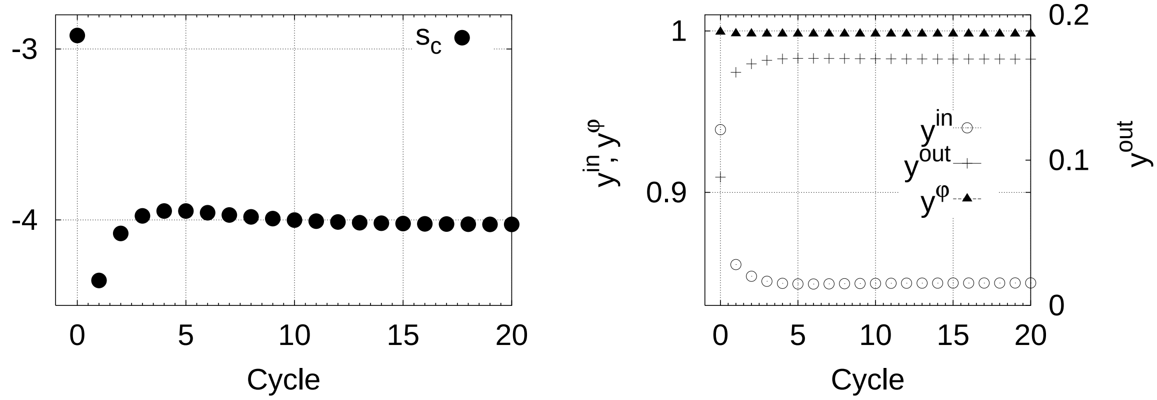

Figure 13: Cell states and gate activations at the onset (zero phase) of the first 20 cycles during a test run with a trained LSTM network on the $f_{\cos}$ PFG task with $F=10$. The initial state (at cycle 0) is quite far from the equilibrium state.

Memory capacity remains a problem for RNNs. For LSTM, memory is limited by the number of memory blocks in the network. It is unlikely that this limitation can be overcome simply by increasing the network size homogenously. Modularization of the network topology—and thus the task—will be necessary to keep learning effective. How such modularization can be learned or performed effectively and how the network modules should be interconnected is not generally clear.

RNNs are also limited in the complexity of the solutions they can learn. LSTM for example easily learns to instantiate a counter (equivalent to a one-symbol stack), and by combining two counters it can even learn a simple context-sensitive language (Gers and Schmidhuber, 2001). This seems to be the current limit though—no RNN has learned to instantiate a stack or a queue for a non-trivial number of symbols (say, a dozen).

Due to the error truncation we employed, the gating curcuitry of the LSTM architecture in its present form is not fully recurrent; the gates therefore depend on (classes of) individual input patterns for their operation. Thus in cases where memorization, recall, or forgetting are to be triggered by specific, temporally extended, possibly noisy input *sequences*, it may become necessary to preprocess the input signal with an appropriate feature detector before passing it on to the LSTM network. We have encountered this limitation when trying to extract prosodic information from speech data (Cummins et al., 1999).

5.3 Conclusion

We have presented a neural network architecture that can learn to recognize and robustly generate precisely timed events separated by significant time lags.

Previous work demonstrated LSTM's advantages over traditional RNNs on tasks involving long time lags between relevant input events. Those tasks, however, did not require the network to extract information conveyed by the duration of intervals between these events. Here we have shown 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.

Since the original development of LSTM our work has concentrated on improving structure and wiring of the nonlinear, multiplicative gates surrounding and protecting LSTM's constant error carousels. The LSTM variant with peepholes and forget gates used in this paper clearly outperforms traditional LSTM, and has become our method of choice for RNN applications. We are using it to find prosody information in speech data (Cummins et al., 1999), and to detect and generate rhythm and music (Eck and Schmidhuber, 2002).

Appendix A. Peephole LSTM with Forget Gates in 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 (19):

$z_{\mathrm{in}_j} = \sum_m w_{\mathrm{in}_j m}\ \hat{y}_m + \sum_{v=1}^{S_j} w_{\mathrm{in}_j c_j^v}\ \hat{s}_{c_j^v}\,; \quad y_{\mathrm{in}_j} = f_{\mathrm{in}_j}(z_{\mathrm{in}_j})\,;$

Step 1b: forget gates (20):

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

Step 1c: CECs, i.e the cell states (1, 4):

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

$z_{c_j^v} = \sum_m w_{c_j^v m}\ \hat{y}_m\,; \quad s_{c_j^v} = y_{\varphi_j}\ \hat{s}_{c_j^v} + y_{\mathrm{in}_j}\ g(z_{c_j^v})\,;$ }

Step 2:

output gate activation (21):

$z_{\mathrm{out}_j} = \sum_m w_{\mathrm{out}_j m}\ \hat{y}_m + \sum_{v=1}^{S_j} w_{\mathrm{out}_j c_j^v}\ s_{c_j^v}\,; \quad y_{\mathrm{out}_j} = f_{\mathrm{out}_j}(z_{\mathrm{out}_j})\,;$

cell outputs (5):

`loop over the` S_j `cells in block` j`, indexed` v { $y_{c_j^v} = y_{\mathrm{out}_j}\ s_{c_j^v}\,;$ }

} `end loop over memory blocks`

output units (7): $z_k = \sum_m w_{km}\ y_m$**;** $y_k = f_k(z_k)$**;**

partial derivatives:

`loop over memory blocks, indexed` j {

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

cells (12), $\left(dS_{cm}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{c_j^v m}}\right)$**:**

$dS_{cm}^{jv} = dS_{cm}^{jv}\ y_{\varphi_j} + g'(z_{c_j^v})\ y_{\mathrm{in}_j}\ \hat{y}_m\,;$

input gates (13, 22), $\left(dS_{\mathrm{in},m}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\mathrm{in}_j m}}\ ,\ dS_{\mathrm{in},c_j^{v'}}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\mathrm{in}_j c_j^{v'}}}\right)$**:**

$dS_{\mathrm{in},m}^{jv} = dS_{\mathrm{in},m}^{jv}\ y_{\varphi_j} + g(z_{c_j^v})\ f'_{\mathrm{in}_j}(z_{\mathrm{in}_j})\ \hat{y}_m\,;$

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

$dS_{\mathrm{in},c_j^{v'}}^{jv} = dS_{\mathrm{in},c_j^{v'}}^{jv}\ y_{\varphi_j} + g(z_{c_j^v})\ f'_{\mathrm{in}_j}(z_{\mathrm{in}_j})\ \hat{s}_c^{v'}\,;$ }

forget gates (14, 23), $\left(dS_{\varphi m}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\varphi_j m}}\ ,\ dS_{\varphi c_j^{v'}}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\varphi_j c_j^{v'}}}\right)$**:**

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

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

$dS_{\varphi c_j^{v'}}^{jv} = dS_{\varphi c_j^{v'}}^{jv}\ y_{\varphi_j} + \hat{s}_{c_j^v}\ f'_{\varphi_j}(z_{\varphi_j})\ \hat{s}_c^{v'}\,;$ }

} } `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 (9): $\delta_k = f'_k(z_k)\ e_k$;

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

δs of output gates (11):

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

internal state error (18):

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

$e_{s_{c_j^v}} = y_{\mathrm{out}_j}\ \left(\sum_k w_{kc_j^v}\ \delta_k\right); \quad \}$

$\}$ `end loop over memory blocks`

weight updates:

output units (8): $\Delta w_{km} = \alpha\ \delta_k\ y_m$;

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

output gates (10):

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

input gates (16):

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

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

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

forget gates (17):

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

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

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

cells (15):

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

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

$\}$ `end loop over memory blocks`

Acknowledgment

This work was supported by the Swiss National Science Foundation under grant number 2100-49'144.96, "Long Short-Term Memory".

References

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

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

A.B. Bulsari and H. Saxén. A recurrent network for modelling noisy temporal sequences. *Neurocomputing*, 7:29–40, 1995.

Fred Cummins, Felix Gers, and Jürgen Schmidhuber. Language identification from prosody without explicit features. In *Proceedings of EUROSPEECH'99*, volume 1, pages 371–374, 1999.

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

D. Eck and J. Schmidhuber. Learning the long-term structure of the blues. In *Proc. Intl. Conf. Artificial Neural Networks*, Lecture Notes in Computer Science. Springer Verlag, Berlin (in press), 2002.

Mikel L. Forcada and Rafael C. Carrasco. Learning the initial state of a second-order recurrent neural network during regular-language inference. *Neural Computation*, 7(5):923–930, 1995.

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

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

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

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

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

Y. LeCun, L. Bottou, G.B. Orr, and K.-R. Müller. Efficient backprop. In Genevieve B. Orr and Klaus-Robert Müller, editors, *Neural Networks—Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 5–50. Springer Verlag, Berlin, 1998.

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

D. C. Plaut, S. J. Nowlan, and G. E. Hinton. Experiments on learning back propagation. Technical Report CMU–CS–86–126, Carnegie–Mellon University, Pittsburgh, PA, 1986.

A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

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

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

S. Townley, A. Ilchmann, M. G. Weiss, W. McClements, A. C. Ruiz, D. Owens, and D. Praetzel-Wolters. Existence and learning of oscillations in recurrent neural networks. Technical Report AGTM 202, Universitaet Kaiserslautern, Fachbereich Mathematik, Kaiserslautern, Germany, 1999.

Fu-Sheng Tsung and Garrison W. Cottrell. Phase-space learning. In *Advances in Neural Information Processing Systems*, volume 7, pages 481–488. The MIT Press, 1995.

M. G. Weiss. Learning oscillations using adaptive control. Technical Report AGTM 178, Universitaet Kaiserslautern, Fachbereich Mathematik, Kaiserslautern, Germany, 1999.

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

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

R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin and D. E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*, chapter 13, pages 433–486. Hillsdale, NJ: Erlbaum, 1992.