Bridging Long Time Lags by Weight Guessing and "Long Short Term Memory"

Sepp Hochreiter
Fakultät für Informatik, Technische Universität München
80290 München, Germany

Jürgen Schmidhuber
IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland

Abstract. Numerous recent papers (including many NIPS papers) focus on standard recurrent nets' inability to deal with long time lags between relevant input signals and teacher signals. Rather sophisticated, alternative methods were proposed. We first show: problems used to promote certain algorithms in numerous previous papers can be solved more quickly by random weight guessing than by the proposed algorithms. This does not mean that guessing is a good algorithm. It just casts doubt on whether the other algorithms are, or whether the chosen problems are meaningful. We then use long short term memory (LSTM), our own recent algorithm, to solve hard problems that can neither be quickly solved by random weight guessing nor by any other recurrent net algorithm we are aware of.

1 Introduction / Outline

Many recent papers focus on standard recurrent nets' inability to deal with long time lags between relevant signals. See, e.g., Bengio et al., El Hihi and Bengio, and others [3, 1, 6, 15]. Rather sophisticated, alternative methods were proposed. For instance, Bengio et al. investigate methods such as simulated annealing, multi-grid random search, time-weighted pseudo-Newton optimization, and discrete error propagation [3]. They also propose an EM approach for propagating targets [1].

Quite a few papers use Bengio et al.'s so-called "latch problem" (and "2-sequence problem") to show the proposed algorithms's superiority, e.g. [3, 1, 6, 15]. For the same purpose, some papers also use the so-called "parity problem", e.g., [3, 1]. Some of Tomita's grammars [30] are also often used as benchmark problems for recurrent nets [2, 31, 23, 17]. We will show: all these and similar problems are not very useful to promote the proposed/investigated algorithms, because they turn out to be trivial. They can be solved more quickly by random weight guessing than by the proposed algorithms [27]. We argue that novel long time lag algorithms should be tested on non-trivial problems whose solutions cannot be quickly guessed. We then show: LSTM, our own recent algorithm, can solve hard problems that can neither be solved by random weight guessing nor by any other recurrent net algorithm we are aware of. See [13] for a variant of this paper.

2 The 2-Sequence Problem and the Parity Problem Etc. are Trivial

Trivial versus non-trivial tasks. By our definition, a "trivial" task is one that can be solved quickly by random search in weight space. Random search works as follows: *REPEAT randomly initialize the weights and test the resulting net on a training set UNTIL solution found.*

Random search details. In all our experiments, we randomly initialize weights in [-100.0,100.0]. Binary inputs are -1.0 (for 0) and 1.0 (for 1). Targets are either 1.0 or 0.0. All activation functions are logistic and sigmoid in [0.0,1.0]. We use two architectures (A1, A2) suitable for many widely used "benchmark" problems: A1 is a fully connected net with 1 input, 1 output, and n biased hidden units. A2 is like A1 with $n = 10$, but somewhat less densely connected: each hidden unit sees the input unit, the output unit, and itself; the output unit sees all other units; all units are biased. All activations are set to 0 at each sequence begin. We will indicate where we also use different architectures of other authors. All sequence lengths are randomly chosen between 500 and 600 (most other authors facilitate their problems by using much shorter training/test sequences). The "benchmark" problems always require to classify two types of sequences. Our training set consists of 100 sequences, 50 from class 1 (target 0) and 50 from class 2 (target 1). Correct sequence classification is defined as "absolute error at sequence end below 0.1". We stop the search once a random weight matrix correctly classifies all training sequences. Then we test on the test set (100 sequences). All results below are averages of 10 trials. *In all our simulations below, guessing finally classified all test set sequences correctly; average final absolute test set errors were always below 0.001 — in most cases below 0.0001.*

"2-sequence problem" (and "latch problem") [3, 1, 15]. The task is to observe and classify input sequences. There are two classes. There is only one input unit or input line. Only the first N real-valued sequence elements convey relevant information about the class. Sequence elements at positions $t > N$ (we use $N = 1$) are generated by a Gaussian with mean zero and variance 0.2. The first sequence element is 1.0 (-1.0) for class 1 (2). Target at sequence end is 1.0 (0.0) for class 1 (2) (the latch problem is a simple version of the 2-sequence problem that allows for input tuning instead of weight tuning).

Bengio et al.'s results. For the 2-sequence problem, the best method among the six tested by Bengio et al. [3] was multigrid random search (sequence lengths 50 — 100; no precise stopping criterion mentioned), which solved the problem after 6,400 sequence presentations, with final classification error 0.06. In more recent work, Bengio and Frasconi were able to improve their results: an EM-approach [1] was reported to solve the problem within 2,900 trials.

Results with guessing. Random guessing with architecture A2 (A1, $n = 1$) solves the problem within only 718 (1247) trials on average. Using Bengio et al.'s architecture for the latch problem [3] (only 3 parameters), the problem was solved *within only 22 trials on average,* due to tiny parameter space. According to our definition above, the problem is trivial. Random guessing outperforms Bengio et al.'s methods in every respect: (1) many fewer trials required, (2) less computation time per trial. Also, in most cases (3) the solution quality is better (less error).

"Parity problem". Bengio et al.'s (and Bengio and Frasconi's) parity task [3, 1] requires to classify sequences with several 100 elements (only 1's or -1's) according to whether the number of 1's is even or odd. The target at sequence end is 1.0 for odd and 0.0 for even.

Bengio et al.'s results. For sequences with only 25-50 steps, among the six methods tested in [3], only simulated annealing was reported to achieve final classification error of 0.000 (within about 810,000 trials — the authors did not mention the precise stopping criterion). A method called "discrete error BP" took about 54,000 trials to achieve final classification error 0.05. In their more recent work [1], for sequences with 250-500 steps, their EM-approach took about 3,400 trials to achieve final classification error 0.12.

Guessing results. Guessing with A1 ($n = 1$, identical to Bengio et al.'s architecture in [3]) solves the problem within only 2906 trials on average. Guessing with A2 solves it within 2797 trials. We also ran another experiment with architecture A2, but without self-connections for hidden units. Guessing solved the problem within 250 trials on average.

Tomita grammars. Many authors also use Tomita's grammars [30] to test their algorithms. See, e.g., [2, 32, 22, 17, 16]. Since we already tested parity problems above, we now focus on a few "parity-free" Tomita grammars (nr.s #1, #2, #4). Previous work even facilitated the problems by restricting sequence length. E.g., in [17], maximal test (training) sequence length is 15 (10). Reference [17] reports the number of sequences required for convergence (for various first and second order nets with 3 to 9 units): Tomita #1: 23,000 – 46,000; Tomita #2: 77,000 – 200,000; Tomita #4: 46,000 – 210,000. Random guessing, however, clearly outperforms the methods in [17]. The average results are: Tomita #1: 182 (A1, $n = 1$) and 288 (A2), Tomita #2: 1,511 (A1, $n = 3$) and 17,953 (A2), Tomita #4: 13,833 (A1, $n = 2$) and 35,610 (A2).

Flat minima. It should be mentioned that successful guessing typically hits flat minima of the error function [11].

Non-trivial tasks / Outline of remainder. We believe that novel long time lag algorithms should be tested on problems whose solutions cannot be quickly guessed. The experiments in the remainder of this paper deal with non-trivial tasks whose solutions are sparse in weight space. They require either many free parameters (e.g., input weights) or high weight precision, such that random guessing becomes completely infeasible. All experiments involve long minimal time lags — there are no short time lag training exemplars facilitating learning. To solve these tasks, however, we need the novel method called "Long Short Term Memory", or LSTM for short [10]. Section 3 will briefly review LSTM. Section 4 will present new results on tasks that cannot be solved at all by any other recurrent net learning algorithm we are aware of. LSTM can solve rather complex long time lag problems involving distributed, high-precision, continuous-valued representations, and is able to extract information conveyed by the temporal order of widely separated inputs.

3 Long Short Term Memory

Memory cells and gate units: basic ideas. LSTM's basic unit is called a memory cell. Within each memory cell, there is a linear unit with a fixed-weight self-connection (compare Mozer's time constants [19]). This enforces constant, non-exploding, non-vanishing error flow within the memory cell. A multiplicative *input gate unit* learns to protect the constant error flow within the memory cell from perturbation by irrelevant inputs. Likewise, a multiplicative *output gate unit* learns to protect other units from perturbation by currently irrelevant memory contents stored in the memory cell. The gates learn to open and close access to constant error flow. *Why is constant error flow important?* For instance, with conventional "backprop through time" (BPTT, e.g., [33]) or RTRL (e.g., [25]), error signals "flowing backwards in time" tend to either (1) blow up or (2) vanish: the temporal evolution of the backpropagated error exponentially depends on the size of the weights. Case (1) may lead to oscillating weights. In case (2), learning to bridge long time lags takes a prohibitive amount of time, or does not work at all — for a detailed theoretical analysis of error blow-ups/vanishing errors, see [9] (the vanishing error case was later also treated in [3]).

LSTM Details. In what follows, w_{uv} denotes the weight on the connection from unit v to unit u. $net_u(t), y^u(t)$ are net input and activation of unit u (with activation function f_u) at time t. For all non-input units that aren't memory cells (e.g. output units), we have $y^u(t) = f_u(net_u(t))$, where $net_u(t) = \sum_v w_{uv} y^v(t-1)$. The j-th memory cell is denoted c_j. Each memory cell is built around a central linear unit with a fixed self-connection (weight 1.0) and identity function as activation function (see definition of s_{c_j} below). In addition to $net_{c_j}(t) = \sum_u w_{c_j u} y^u(t-1)$, c_j also gets input from a special unit out_j (the "output gate"), and from another special unit in_j (the "input gate"). in_j's activation at time t is denoted by $y^{in_j}(t)$. out_j's activation at time t is denoted by $y^{out_j}(t)$. We have $y^{out_j}(t) = f_{out_j}(net_{out_j}(t)), y^{in_j}(t) = f_{in_j}(net_{in_j}(t))$, where $net_{out_j}(t) = \sum_u w_{out_j u} y^u(t-1)$, $net_{in_j}(t) = \sum_u w_{in_j u} y^u(t-1)$. The summation indices u may stand for input units, gate units, memory cells, or even conventional hidden units if there are any (see also paragraph on "network topology" below). All these different types of units may convey useful information about the current state of the net. For instance, an input gate (output gate) may use inputs from other memory cells to decide whether to store (access) certain information in its memory cell. There even may be recurrent self-connections like $w_{c_j c_j}$. It is up to the user to define the network topology. At time t, c_j's output $y^{c_j}(t)$ is computed in a sigma-pi-like fashion:

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

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

Why gate units? in_j controls the error flow to memory cell c_j's input connections $w_{c_j u}$. The net can use in_j to decide when to keep or override information in memory cell c_j. out_j controls the error flow from unit j's output connections. The net can use out_j to decide when to access memory cell c_j and when to prevent other units from being perturbed by c_j. Error signals trapped within a memory cell *cannot* change – but different error signals flowing into the cell (at different times) via its output gate may get superimposed. The output gate will have to learn *which* errors to trap in its memory cell, by appropriately scaling them. Likewise, the input gate will have to learn when to release errors. The gate units open and close access to constant error flow.

Network topology. There is one input layer, one hidden layer, and one output layer. The fully self-connected hidden layer contains memory cells and corresponding gate units. The hidden layer may also contain "conventional" hidden units providing inputs to gate units and memory cells. All units (except for gate units) in all layers have directed connections (serve as inputs) to all units in higher layers.

Memory cell blocks. S memory cells sharing one input gate and one output gate form a "memory cell block of size S". Memory cell blocks facilitate information storage — like with conventional neural nets, it is not so easy to code multiple distributed inputs within a single cell.

Learning with excellent computational complexity — see details in appendix of [12]. We use a variant of RTRL which properly takes into account the altered (sigma-pi-like) dynamics caused by input and output gates. However, to ensure constant error backprop, like with truncated BPTT [33], errors arriving at "memory cell net inputs" (for cell c_j, this includes net_{c_j}, net_{in_j}, net_{out_j}) do not get propagated back further in time (although they *do* serve to change the incoming weights). Only within memory cells, errors are propagated back through previous internal states s_{c_j}. This enforces constant error flow within memory cells. Thus, like with Mozer's focused recurrent backprop algorithm [18], only the derivatives $\frac{\partial s_{c_j}}{\partial w_{il}}$ need to be stored and updated. Hence, *the algorithm is very efficient*, and LSTM's update complexity per time step is excellent in comparison to other approaches such as RTRL: given n units and a fixed number of output units, LSTM's update complexity per time step is at most $O(n^2)$, just like BPTT's.

4 Experiments

Our previous experimental comparisons [10] (on widely used benchmark problems) with "Real-Time Recurrent Learning" (RTRL, e.g., [25]; results compared to the ones in [28]), "Recurrent Cascade-Correlation" [8], "Elman nets", (results compared to the ones in [4]), and "Neural Sequence Chunking" [26], already demonstrated that LSTM leads to many more successful runs than its competitors, and learns much faster [10]. The following tasks, though, are more difficult than the above benchmark problems: they cannot be solved at all in reasonable time by random search (we tried various architectures) nor any other recurrent net learning algorithm we are aware of, e.g., [25, 20, 7, 8, 33, 14, 21, 5, 15, 29, 1, 24, 26, 19].

In the experiments below, gate units (f_{in_j}, f_{out_j}) and output units are sigmoid in $[0,1]$. h is sigmoid in $[-1,1]$, and g is sigmoid in $[-2,2]$. Weights are initialized in $[-0.1,0.1]$. All non-input units are biased, and the output layer receives connections from memory cells only. Memory cells/gate units receive inputs from input units, memory cells, gate units (fully connected hidden layer — less connectivity works as well). Error signals occur only at sequence ends.

4.1 Experiment 1: Adding Problem

The experiment will show that LSTM can solve non-trivial, complex long time lag problems involving distributed, high-precision, continuous-valued representations.

Task. Each element of each input sequence is a pair consisting of two components. The first component is a real value randomly chosen from the interval $[-1,1]$. The second component is either 1.0, 0.0, or -1.0, and is used as a marker: at the end of each sequence, the task is to output the sum of the first components of those pairs that are *marked* by second components equal to 1.0. The value T is used to determine average sequence length, which is a randomly chosen integer between T and $T+\frac{T}{10}$. With a given sequence, exactly two pairs are marked as follows: we first randomly select and mark one of the first ten pairs (whose first component is called X_1). Then we randomly select and mark one of the first $\frac{T}{2}-1$ still unmarked pairs (whose first component is called X_2). The second components of the remaining pairs are zero except for the first and final pair, whose second components are -1 (X_1 is set to zero in the rare case where the *first* pair of the sequence got marked). An error signal is generated only at the sequence end: the target is $0.5+\frac{X_1+X_2}{4.0}$ (the sum X_1+X_2 scaled to the interval $[0,1]$). A sequence was processed correctly if the absolute error at the sequence end is below 0.04.

Architecture. We use a 3-layer net with 2 input units, 1 output unit, and 2 memory cell blocks of size 2 (a cell block size of 1 works well, too). The output layer receives connections only from memory cells. Memory cells/gate units receive inputs from memory cells/gate units (fully connected hidden layer — less connectivity may work as well).

State drift versus initial bias. Note that the task requires to store the precise values of real numbers for long durations — the system must learn to protect memory cell contents against even minor "internal state drifts". Our simple but highly effective way of solving drift problems at the beginning of learning is to initially bias the input gate in_j towards zero. *There is no need for fine tuning initial bias:* with sigmoid logistic activation functions, the precise initial bias hardly matters because vastly different initial bias values produce almost the same near-zero activations. In fact, the system itself learns to generate the most appropriate input gate bias. To study the significance of the drift problem, we bias all non-input units, thus artificially inducing internal state drifts. Weights (including bias weights) are randomly initialized in the range $[-0.1,0.1]$. The first (second) input gate bias is initialized with -3.0 (-6.0) (recall that the precise initialization values hardly matters, as confirmed by additional experiments).

Training / Testing. The learning rate is 0.5. Training examples are generated on-line. Training is stopped if the average training error is below 0.01, and the 2000 most recent sequences were processed correctly (see definition above).

Results. With a test set consisting of 2560 randomly chosen sequences, the average test set error was always below 0.01, and there were never more than 3 incorrectly processed sequences. The following results are means of 10 trials: For $T=100$ ($T=500$, $T=1000$), training was stopped after 74,000 (209,000; 853,000) training sequences, and then only 1 (0, 1) of the test sequences was not processed correctly. For $T=1000$, the number of required training examples varied between 370,000 and 2,020,000, exceeding 700,000 in only 3 cases.

The experiment demonstrates even for very long time lags: (1) LSTM is able to work well with distributed representations. (2) LSTM is able to perform calculations involving *high-precision, continuous* values. Such tasks are impossible to solve within reasonable time by other algorithms: the main problem of gradient-based approaches (including TDNN, pseudo Newton) is their inability to deal with very long minimal time lags (vanishing gradient). A main problem of "global" and "discrete" approaches (random search, Bengio's and Frasconi's EM-approach, discrete error propagation) is their inability to deal with high-precision, continuous values.

4.2 Experiment 2: Temporal Order

In this subsection, LSTM solves another task that cannot be solved at all by any other recurrent net learning algorithm we are aware of. The task was suggested by Mike Mozer (personal communication).

Task 2a: two relevant, widely separated symbols. The goal is to classify sequences. Elements are represented locally (binary input vectors with only one non-zero bit). The 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 two elements at positions t_1 and t_2 that are either X or Y. The sequence length is randomly chosen between 100 and 110, t_1 is randomly chosen between 10 and 20, and t_2 is randomly chosen between 50 and 60. There are 4 sequence classes Q, R, S, U which depend on the temporal order of X and Y. The rules are: $X, X \rightarrow Q$; $X, Y \rightarrow R$; $Y, X \rightarrow S$; $Y, Y \rightarrow U$.

Task 2b: three relevant, widely separated symbols. Again, the goal is to classify sequences. Elements are represented locally. The 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. 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_2 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: $X, X, X \rightarrow Q$; $X, X, Y \rightarrow R$; $X, Y, X \rightarrow S$; $X, Y, Y \rightarrow U$; $Y, X, X \rightarrow V$; $Y, X, Y \rightarrow A$; $Y, Y, X \rightarrow B$; $Y, Y, Y \rightarrow C$.

With both tasks, error signals occur only at the end of a sequence. The sequence is classified correctly if the final error of all output units is below 0.3.

Architecture. We use a 3-layer net with 8 input units, 2 (3) cell blocks of size 2 for task 2a (2b), 4 (8) output units for task 2a (2b). Again, non-input units are biased, and the output layer receives connections from memory cells only. Memory cells/gate units receive inputs from input units, memory cells, gate units (fully connected hidden layer — less connectivity works as well).

Training / Testing. The learning rate is 0.5 (0.1) for experiment 2a (2b). Training examples are generated on-line. Training is stopped if average training error is below 0.1, and the 2000 most recent sequences were classified correctly. Weights are initialized in $[-0.1, 0.1]$. The first (second) input gate bias is initialized with -2.0 (-4.0) (again, precise initialization values hardly matter, as confirmed by additional experiments).

Results. With a test set consisting of 2560 randomly chosen sequences, the average test set error was always below 0.1, and there were never more than 3 incorrectly classified sequences. The following results are means of 20 trials: For task 2a (2b), training was stopped (see stopping criterion in previous paragraph) after on average 31,390 (571,100) training sequences, and then only 1 (2) of the 2560 test sequences were not classified correctly (see definition above). Obviously, LSTM is able to extract information conveyed by the temporal order of widely separated inputs.

Conclusion. For non-trivial tasks (where random weight guessing is infeasible), we recommend LSTM.

5 ACKNOWLEDGMENTS

This work was supported by *DFG grant SCHM 942/3-1* from "Deutsche Forschungsgemeinschaft".

References

[1] Y. Bengio and P. Frasconi. Credit assignment through time: Alternatives to backpropagation. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 75–82. San Mateo, CA: Morgan Kaufmann, 1994.

[2] Y. Bengio and P. Frasconi. An input output HMM architecture. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 427–434. MIT Press, Cambridge MA, 1995.

[3] 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.

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

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

[6] S. El Hihi and Y. Bengio. Hierarchical recurrent neural networks for long-term dependencies. In *Advances in Neural Information Processing Systems 8*, 1995. To appear.

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

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

[9] J. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[10] S. Hochreiter and J. Schmidhuber. Long short term memory. Technical Report FKI-207-95, Fakultät für Informatik, Technische Universität München, 1995.

[11] S. Hochreiter and J. Schmidhuber. Flat minima. *Neural Computation*, 1996. In press. Extended version available in WWW homepages of Hochreiter and Schmidhuber.

[12] S. Hochreiter and J. Schmidhuber. Long short term memory. *Submitted to Neural Computation*, 1996.

[13] S. Hochreiter and J. Schmidhuber. LSTM can solve hard long time lag problems. 1996. Submitted to NIPS 96.

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

[15] T. Lin, B. G. Horne, P. Tino, and C. L. Giles. Learning long-term dependencies is not as difficult with NARX recurrent neural networks. Technical Report UMIACS-TR-95-78 and CS-TR-3500, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, 1995.

[16] P. Manolios and R. Fanelli. First-order recurrent neural networks and deterministic finite state automata. *Neural Computation*, 6:1155–1173, 1994.

[17] C. B. Miller and C. L. Giles. Experimental comparison of the effect of order in recurrent neural networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(4):849–872, 1993.

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

[19] M. C. Mozer. Induction of multiscale temporal structure. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 4*, pages 275–282. San Mateo, CA: Morgan Kaufmann, 1992.

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

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

[22] J. B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:227–252, 1991.

[23] J. B. Pollack. Language induction by phase transition in dynamical recognizers. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 619–626. San Mateo, CA: Morgan Kaufmann, 1991.

[24] M. B. Ring. Learning sequential tasks by incrementally adding higher orders. In J. D. Cowan S. J. Hanson and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 115–122. Morgan Kaufmann, 1993.

[25] 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.

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

[27] J. Schmidhuber and S. Hochreiter. Guessing can outperform many long time lag algorithms. Technical Report IDSIA-19-96, IDSIA, 1996.

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

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

[30] M. Tomita. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Cognitive Science Conference*, pages 105–108. Ann Arbor, MI, 1982.

[31] R. L. Watrous and G. M. Kuhn. Induction of finite-state automata using second-order recurrent networks. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 309–316. San Mateo, CA: Morgan Kaufmann, 1992.

[32] R. L. Watrous and G. M. Kuhn. Induction of finite-state languages using second-order recurrent networks. *Neural Computation*, 4:406–414, 1992.

[33] R. J. Williams and J. Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 4:491–501, 1990.