Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks *

C.L. Giles[1], C.B. Miller
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
E-mail: giles@research.nj.nec.com

D. Chen, H.H. Chen, G.Z. Sun, Y.C. Lee
University of Maryland
[1]Institute for Advanced Computer Studies
Department of Physics and Astronomy
College Park, Md. 20742

Abstract

We show that a recurrent, second-order neural network using a real-time, forward training algorithm readily learns to infer small regular grammars from positive and negative string training samples. We present simulations which show the effect of initial conditions, training set size and order, and neural network architecture. All simulations were performed with random initial weight strengths and usually converge after approximately a hundred epochs of training. We discuss a quantization algorithm for dynamically extracting finite state automata during and after training. For a well-trained neural net, the it extracted automata constitute an equivalence class of state machines that are reducible to the minimal machine of the inferred grammar. We then show through simulations that many of the neural net state machines are dynamically

*Published in *Neural Computation* 4, 393-405, 1992

stable, i.e. they correctly classify many long unseen strings. In addition, some of these extracted automata actually outperform the trained neural network for classification of unseen strings.

1 INTRODUCTION

Grammatical inference, the problem of inferring grammar(s) from sample strings of a language, is a hard problem, even for regular grammars; for a discussion of the levels of difficulty see [Gold 78] and [Angluin 83]. Consequently, there have been many heuristic algorithms developed for grammatical inference; which either scale poorly with the number of states of the inferred automata or require additional information such as restrictions on the type of grammar or the use of queries [Angluin 83]. For a summary of inference methods, see [Fu 82], [Angluin 83] and the recent, comprehensive summary by [Miclet 90].

The history of finite state automata and neural networks is a long one. For example, [Minsky 67] proved that 'Every finite-state machine is equivalent to, and can be simulated by some neural network.' More recently the training of first-order recurrent neural networks that recognize finite state languages was discussed by [Williams 89], [Cleeremans 89] and [Elman 90]. The recurrent networks were trained by predicting the next symbol and using a truncation of the backward recurrence. [Cleeremans 89] concluded that the hidden unit activations represented past histories and that clusters of these activations can represent the states of the generating automaton. [Mozer 90] applies a neural network approach with a second-order gating term to a query learning method [Rivest 87]. These methods ([Rivest 87] and [Mozer 90]) require *active exploration* of the unknown environments, and produce very good finite state automata (FSA) models of those environments.

We discuss a recurrent neural network solution to grammatical inference and show that second-order recurrent neural networks learn fairly well small regular grammars with an infinite number of strings. This greatly expands upon our previous work [Giles 90] and [Liu 90] which only considered regular grammars of unusual state symmetries. Our approach is similar to that of [Pollack 90] and differs in the learning algorithm (the gradient computation is not truncated) and the emphasis on what is to be learned. In contrast to [Pollack 90], we emphasize that a recurrent network can be trained to exhibit fixed-point behavior and correctly classify long, previously unseen, strings. [Watrous 92] illustrates similar results using another complete gradient calculation method. We also show that from different trained neural networks, a large equivalence class of FSA can be extracted. This is an important extension of the work of [Cleeremans 89] where only the states of the FSA where extracted. This work illustrates a method that permits not only the extraction of the states of the FSA, but the *full* FSA itself.

2 GRAMMARS

2.1 Formal Grammars and Grammatical Inference

We give a brief introduction to formal grammars and grammatical inference; for a thorough introduction, we recommend respectively, [Harrison 78] and [Fu 82]. Briefly, a grammar G is a four tuple {N,T,P,S}, where N and T are sets of nonterminals and terminals (alphabet of the grammar), P a set of production rules and S the start symbol. For every grammar, there exists a language L, a set of strings of the terminal symbols, that the grammar generates or recognizes. There also exist automata which recognize and generate that grammar. In the Chomsky hierarchy of phrase structured grammars, the simplest grammar and its associated automata are regular grammars and finite state automata (FSA). This is the class of grammars we will discuss here. It is important to realize that all grammars whose string length and alphabet size are bounded are regular grammars and can be recognized and generated, maybe inefficiently, by finite state automata.

Grammatical inference is concerned mainly with the procedures that can be used to infer the syntactic rules (or production rules) of an unknown grammar G based on a finite set of strings $\mathcal{I}$ from L(G), the language generated by G and possibly also on a finite set of strings from the complement of L(G) [Fu 82]. Positive examples of the input strings are denoted as $\mathcal{I}_+$ and negative examples as $\mathcal{I}_-$. We replace the inference algorithm with a recurrent second-order neural network, and the training set consists of both positive and negative strings.

2.2 Grammars of Interest

In order to explore the inference capabilities of the recurrent neural net, we have chosen to study a set of seven relatively simple grammars originally created and studied by [Tomita 82] and recently by [Pollack 90], [Giles 91], and [Watrous 92]. We hypothesize that formal grammars are excellent learning benchmarks; that no feature extraction is required since the grammar itself constitutes the most primitive representation. For very complex grammars, such as the regular grammar that represents Rubik's cube, the feature extraction hypothesis might break down and some feature extraction method, such as *diversity* [Rivest 87], would be necessary. The grammars shown here are simple regular grammars and should be learnable. They all generate infinite languages over {0,1}* and are represented by finite state automata of between three and six states. Briefly, the languages these grammars generate can be described as follows:

#1 — 1*,

#2 — (1 0)*,

#3 — an odd number of consecutive 1's is always followed by an even number of consecutive 0's,

#4 — any string not containing "000" as a substring,

#5 — even number of 0's and even number of 1's, (*see p383, [Giles 90]*), [our interpretation of Tomita #5],

#6 — number of 1's - number of 0's is a multiple of 3,

#7 — 0* 1* 0* 1*.

The FSA for Tomita grammar #4 is given in figure 1c. Note that this FSA contains a so-called "garbage state", that is, a non-final state in which all transition paths lead back to the same state. This means that the recurrent neural net must not only learn the grammar but also its complement and thus correctly classify negative examples. Not all FSA will have garbage states. In this case there are no situations where "illegal characters" occur – there are no identifiable substrings which could independently cause a string to be rejected.

3 RECURRENT NEURAL NETWORK

3.1 Architecture

Recurrent neural networks have been shown to have powerful capabilities for modeling many computational structures; an excellent discussion of recurrent neural network models and references can be found in [Hertz 91]. To learn grammars, we use a second-order recurrent neural network [Lee 86], [Giles 90], [Sun 90], [Pollack 90]. This net has N recurrent hidden neurons labeled S_j; L special, nonrecurrent input neurons labeled I_k; and $N^2 \times L$

real-valued weights labeled W_{ijk}. As long as the number of input neurons is small compared to hidden neurons, the complexity of the network only grows as $O(N^2)$, the same as a linear network. We refer to the values of the hidden neurons collectively as a state *vector* **S** in the finite N-dimensional space $[0,1]^N$. Note that the weights W_{ijk} modify a product of the hidden S_j and input I_k neurons. This quadratic form directly represents the state transition diagrams of a state process — $\{input, state\} \Rightarrow \{nextstate\}$. This recurrent network accepts a time-ordered sequence of inputs and evolves with dynamics defined by the following equations:

$$S_i^{(t+1)} = g(\Xi_i), \qquad \Xi_i \equiv \sum_{j,k} W_{ijk} S_j^{(t)} I_k^{(t)},$$

where g is a sigmoid discriminant function. Each input string is encoded into the input neurons one character per discrete time step t. The above equation is then evaluated for each hidden neuron S_i to compute the next state vector **S** of the hidden neurons at the next time step $t+1$. With unary encoding the neural network is constructed with one input neuron for each character in the alphabet of the relevant language. This condition might be restrictive for grammars with large alphabets.

3.2 Training Procedure

For any training procedure, one must consider the error criteria, the method by which errors change the learning process, and the presentation of the training samples. The error function E_0 is defined by selecting a special "response" neuron S_0 which is either on ($S_0 > 1-\epsilon$) if an input string is accepted, or off ($S_0 < \epsilon$) if rejected, where ϵ is the response tolerance of the response neuron. We define two error cases: (1) the network fails to reject a negative string $\mathcal{I}_-$ (*i.e.* $S_0 > \epsilon$); (2) the network fails to accept a positive string $\mathcal{I}_+$ (*i.e.*

$S_0 < 1 - \epsilon$). For these studies, the acceptance or rejection of an input string is determined only at the end of the presentation of *each* string. The error function is defined as:

$$E_0 = \frac{1}{2}(\tau_0 - S_0^{(f)})^2,$$

where τ_0 is the desired or *target* response value for the response neuron S_0. The target response is defined as $\tau_0 = 0.8$ for positive examples and $\tau_0 = 0.2$ for negative. The notation $S_0^{(f)}$ indicates the *final* value of S_0, *i.e.*, after the final input symbol.

The training is an on-line (real-time) algorithm that updates the weights at the end of each sample string presentation (assuming there is an error $E_0 > .5\epsilon^2$) with a gradient-descent weight update rule:

$$\Delta W_{lmn} = -\alpha \frac{\partial E_0}{\partial W_{lmn}} = \alpha(\tau_0 - S_0^{(f)}) \cdot \frac{\partial S_0^{(f)}}{\partial W_{lmn}},$$

where α is the learning rate. We also add a momentum term, an additive update to ΔW_{lmn}, which is η, the momentum, times the previous ΔW_{lmn}. To determine ΔW_{lmn}, the $\partial S_i^{(f)} / \partial W_{lmn}$ must be evaluated. From the recursive network state equation, we see that

$$\frac{\partial S_i^{(f)}}{\partial W_{lmn}} = g'(\Xi_i) \cdot \left[\delta_{il} S_m^{(f-1)} I_n^{(f-1)} + \sum_{j,k} W_{ijk} I_k^{(f-1)} \frac{\partial S_j^{(f-1)}}{\partial W_{lmn}} \right],$$

where g' is the derivative of the discriminant function. In general, f and $f-1$ can be replaced by any t and $t-1$, respectively. These partial derivative terms are calculated iteratively as the equation suggests, with one iteration per input symbol. This on-line learning rule is a second order form of the recurrent net of [Williams 89]. The initial terms $\partial S_i^{(0)} / \partial W_{lmn}$ are set to zero. After the choice of the initial weight values, the $\partial S_i^{(t)} / \partial W_{lmn}$ can be evaluated *in real time* as each input $I_k^{(t)}$ enters the network. In this way, the error term is forward-propagated and accumulated at each time step t. However, each update of

$\partial S_i^{(t)}/\partial W_{lmn}$ requires $O(N^4 \times L^2)$ terms. For $N >> L$, this update is $O(N^4)$ which is the same as a linear network. This could seriously prohibit the size of the recurrent net if it remains fully interconnected.

3.3 Presentation of Training Samples

The training data consists of a series of stimulus-response pairs, where the stimulus is a string over $\{0,1\}^*$, and the response is either "1" for positive examples or "0" for negative examples. The positive and negative strings, $\mathcal{I}_+$ and $\mathcal{I}_-$, are generated by a source grammar prior to training. Recall that at each discrete time step, one symbol from the string is presented to the neural network. There was *no* total error accumulation as occurs in batch learning; training occurred after each string presentation.

The sequence of strings during training may be very important. In order to avoid too much bias (such as short versus long, positive versus negative), we randomly chose the initial training set of 1024 strings, including Tomita's original set, from the set of all strings of length less than 16 (65,535 strings). As the network starts training, the network only gets to see some small randomly-selected fraction of the training data, about 30 strings. The remaining portion of the data is called "pre-test" training data, which the network gets to see only after it either classifies all 30 examples correctly (*i.e.*, for all strings $|E| < \epsilon$), or reaches a maximum number of epochs (one epoch = the period during which the network processes each string once). This total maximum number of epochs is 5000 and is set before training. When either of these conditions is met, the network checks the pre-test data. The network may add up to 10 misclassified strings in the pre-test data. This prevents the training procedure from driving the network too far towards any local minima that the misclassified strings may represent. Another cycle of epoch training begins with the augmented training set. If the net correctly classifies all the training data, the net is said to *converge*. This is a rather strict sense of convergence. The total number of cycles that the network is permitted to run is also limited, usually to about 20.

An extra *end* symbol is added to the string alphabet to give the network more power in deciding the best final state **S** configuration. For encoding purposes this symbol is simply considered as another character and requires another input neuron. Not that this does not increase the complexity of the FSA! In the training data, the end symbol appears only at the end of each string.

3.4 Extracting State Machines

As the network is training (or after training), we apply a procedure for extracting what the network has learned — *i.e.*, the network's current conception of the FSA it is learning (or has learned). The FSA extraction process includes the following steps: 1) clustering of FSA states, 2) constructing a transition diagram by connecting these states together with the alphabet labelled arcs, 3) putting these transitions together to make the full digraph - forming loops, and 4) reducing the digraph to a minimal representation. The hypothesis is that during training, the network begins to partition (or quantize) its state space into fairly well-separated, distinct regions or clusters, which represent corresponding states in some finite state automaton (see Figure 1). See [Cleeremans 89] for another clustering method. One simple way of finding these clusters is to divide each neuron's range [0,1] into q partitions of equal width. Thus for N hidden neurons, there exist q^N possible *partition states*. The FSA is constructed by generating a state transition diagram, i.e. associating an input symbol with the *partition state* it just left and the *partition state* it activates.

The initial *partition state*, or start state of the FSA, is determined from the initial value of $\mathbf{S}^{(t=0)}$. If the next input symbol maps to the same *partition state* value, we assume that a loop is formed. Otherwise, a new state in the FSA is formed. The FSA thus constructed may contain a maximum of q^N states; in practice it is usually much less, since not all *partition states* are reached by $\mathbf{S}^{(t)}$. Eventually this process must terminate since there are only a finite number of partitions available; and, in practice, many of the partitions are never reached. The *derived* FSA can then be reduced to its minimal FSA using standard minimization algorithms [Hopfcroft 79]. [This minimization process does not change the performance of the FSA; the unminimized FSA has same time complexity as the minimized FSA. The process just rids the FSA of redundant, unnecessary states and reduces the space complexity.] The initial value of the partition parameter is $q = 2$ and is increased only if the extracted FSA fails to correctly classify the 1024 training set. It should be noted that this FSA extraction method may be applied to any discrete-time recurrent net, irregardless of order or hidden layers. Of course this simple partitioning or clustering method could prove difficult for large numbers of neurons.

4 RESULTS - SIMULATIONS

At the beginning of each run, the network is initialized with a set of random weights, each weight chosen between [-1.0,1.0]. Unless otherwise noted, each training session has its own unique initial weight conditions. The initial value of the neurons $S_i^{(t=0)} = \delta_{i0}$; though simulations with these values chosen randomly on the interval [0.0,1.0] showed little significant difference in convergence times. For on-line training the initial hidden neuron values are *never* reset. The hidden neuron values update as new inputs are seen and when weights change due to string misclassification.

The simulations shown in Table 1 focus on Tomita's grammar #4. However, our studies of Tomita's other grammars suggest that the results presented are quite general and apply to any grammar of comparable complexity. Column 1 is a run identification number. The only variables were the number of hidden neurons (3,4,5), in column 2, and the unique random initial weights. In column 3 the initial training set size is 32. Column 4 contains the initial training set size plus all later errors made on the training set. The number of epochs for training is shown in Column 5. If the network doesn't converge in 5000 epochs, we say the network has *failed* to converge on a grammar. The number of errors in the test set (both positive and negative) consisting of the original universe of all strings up to length 15 (65,535–1024 strings) is shown in columns 6, 6a and 6b, and is a measure of the generalization capacity of the trained net. For columns 6 and 6a the error tolerance ϵ is respectively < 0.2 and < 0.5. As expected if the error tolerance is relaxed, the trained network correctly classifies significantly more of the 65K test set. In column 6b are the number of errors for a randomly chosen set of 850,000 strings of length 16 to 99 with an error tolerance < 0.5. The information related to the extraction of FSA from the *trained* neural network is in columns 7-9. The number of neuron partitions (or quantizations) necessary to obtain the FSA which correctly recognizes the original training set is shown in column 7. The partition parameter q is not unique and all or many values of q will actually produce the same minimal FSA if the grammar is well-learned. The number of states of the unminimized *extracted* FSA is shown in column 8. The *extracted* FSA is minimized and in column 9 the number of states of the minimal extracted FSA is shown.

The minimal FSA for the grammar Tomita #4 is 4 states [see figure 1c] if the empty string is accepted and 5 states if the empty string is rejected. The empty string was not used in the training set; consequently, the neural net did not always learn to accept the empty string. However, it is straightforward to include the empty string in the training set. In Figures 1a and 1b are the extracted FSA for two different successful training trial Runs [#104 and #104b in Table 1.] for a 4-neuron neural network. The only difference between the two trials is the initial weight values. The minimized FSA [Hopfcroft 79] for Figures 1a and 1b is shown in Figure 1c. All states in the minimized FSA are final states with the exception of state 0, which is a garbage state. For both cases and in all trials in Table 1 which converged, the minimized extracted FSA is the same as the minimal FSA of Tomita #4. What is interesting is that some extracted FSA, for example trial Runs #60 and #104e, will correctly classify all unseen strings whereas the trained neural networks, from which the FSA were extracted, will not.

5 CONCLUSIONS

Second-order recurrent neural networks are capable of learning small regular grammars rather easily and generalizing very well on unseen grammatical strings. The training results of these neural networks for small simple grammars is fairly independent of the initial values of the weight space and usually converges using an incremental on-line, forward-propagation, training algorithm. For a well-trained neural net, the generalization performance on long (string lengths < 100) unseen strings can be perfect. A heuristic method was used to *extract* finite state automata (FSA) from the neural network, both *during* and *after* training. (It would be interesting if a neural network could also learn to extract the proper FSA.) Using a standard FSA minimization algorithm, the extracted FSA can be reduced to an it equivalent minimal-state FSA. Note that the minimization procedure only reduces the space complexity of the FSA; the time complexity of the minimized and unminimized FSA remains the same. From the extracted FSA, minimal or not, the production rules of the learned grammar are evident.

There are some interesting aspects to the extracted FSA. Surprisingly, each of the unminimized FSA shown in the table is *unique*, even those with the same number of states (i.e., see Runs #105b,d,i,j). For the simple grammar Tomita#4, nearly all networks converged during training (learned the complete training set). For all cases that converged, it is possible to *extract* state machines that are perfect, i.e. the FSA of the unknown source grammar. For these cases the *minimized, extracted* FSA with the same number of states constitute a large *equivalence class* of neural-net-generated FSA, i.e. all unminimized FSA are equivalent and have the *same performance* on string classification. This *equivalence class* extends across neural networks which vary both in size (number of neurons) and initial conditions. Thus, the *extracted* FSA give some indication of how well the neural network learns the grammar.

In fact, for some of the well-trained neural nets, for example Run #104, all extracted, minimized FSA for a large range of partition parameters (2-50) are the same as the ideal FSA of the source grammar. We speculate that for these well-trained neural nets, the extracted, minimal FSA will be *independent* of the choice of the partition parameter. These perfect FSA *outperform* some of the trained neural networks in correct classification of unseen strings. (By definition, a perfect FSA will correctly classify all unseen strings). This is not surprising due to the possibility of error accumulation as the neural network classifies long unseen strings [Pollack 90]. However, when the neural network has learned the grammar well, its generalization performance is also perfect (for all strings tested). Thus, the neural network can be considered as a tool for extracting a FSA that is representative of the unknown grammar. Once the FSA is extracted, it can be used independently of the trained neural network.

Can we make any arguments regarding neural net capacity and scalability? In our simulations the number of states of the *minimal* FSA that was extracted was comparable to the number of neurons in the network; but the actual *extracted, unminimized* FSA had many more states than neurons. However, for Runs #105e and #104h the neural network actually learned an elegant solution, the perfect FSA of the grammar (no minimization was necessary). The question of FSA state capacity and scalability is unresolved. Further work must show how well these approaches can model grammars with large numbers of states and what FSA state capacity of the neural net is theoretically and experimentally reasonable. How a complete-gradient calculation approach using second-order recurrent networks compares to other gradient-truncation, first-order methods [Cleeremans 89] and [Elman 90] is another open question. Surprisingly, a *simple* clustering approach derives useful and representative FSA from a trained or training neural network.

Acknowledgements

We would like to acknowledge useful discussions with M.W. Goudreau, S.J. Hanson, G.M. Kuhn, J. McClelland, J.B. Pollack, E. Sontag, D.S. Touretzky, and R.L. Watrous. The University of Maryland authors gratefully acknowledge partial support through grants from AFOSR and DARPA.

References

[Angluin 83] D. Angluin, C.H. Smith, Inductive Inference: Theory and Methods, *ACM Computing Surveys*, Vol.15, No.3, p.237 (1983).

[Cleeremans 89] A. Cleeremans, D. Servan-Schreiber and J. McClelland, Finite State Automata and Simple Recurrent Recurrent Networks, *Neural Computation*, vol 1, No. 3, p. 372 (1989).

[Elman 90] J.L. Elman, Finding Structure in Time, *Cognitive Science*, vol 14, p. 179 (1990).

[Fu 82] K.S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice-Hall, Englewood Cliffs, N.J. (1982).

[Giles 90] C.L. Giles, G.Z. Sun, H.H. Chen, Y.C. Lee, D. Chen, Higher Order Recurrent Networks & Grammatical Inference, *Advances in Neural Information Systems 2*, D.S. Touretzky (ed), Morgan Kaufmann, San Mateo, Ca, p.380 (1990).

[Giles 91] C.L. Giles, D. Chen, C.B. Miller, H.H. Chen, G.Z. Sun, Y.C. Lee, Grammatical Inference Using Second- Order Recurrent Neural Networks, it Proceedings of the International Joint Conference on Neural Networks IEEE 91CH3049-4, Vol 2, p.357 (1991).

[Gold 78] E.M. Gold, Complexity of Automaton Identification from Given Data, *Information and Control*, Vol.37, p.302 (1978).

[Harrison 78] M.H. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Mass., (1978).

[Hertz 91] J. Hertz, A. Krogh, R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City, Ca., p.163 (1991).

[Hopfcroft 79] J.E. Hopfcroft & J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Mass., p.68 (1979).

[Lee 86] Y.C. Lee, G. Doolen, H.H. Chen, G.Z. Sun, T. Maxwell, H.Y. Lee, C.L. Giles, Machine Learning Using a Higher Order Correlational Network, Physica D, Vol.22-D, No.1-3, p. 276 (1986).

[Liu 90] Y.D. Lu, G.Z. Sun, H.H. Chen, Y.C. Lee, C.L. Giles, Grammatical Inference and Neural Network State Machines, *Proceedings of the International Joint Conference on Neural Networks*, IJCNN-90-WASH-DC, Lawrence Erlbaum, Hillsdale, N.J., Vol I, p.285 (1990).

[Miclet 90] L. Miclet, Grammatical Inference, *Syntactic and Structural Pattern Recognition Theory and Applications*, H. Bunke and A. Sanfeliu (eds), World Scientific, Singapore, Ch 9, (1990).

[Minsky 67] M.L. Minsky, *Computation: Finite and Infinite Machines*, Ch 3.5, Prentice-Hall, Englewood Cliffs, N.J. (1967).

[Mozer 90] M.C. Mozer, J. Bachrach, Discovering the Structure of a Reactive Environment by Exploration, *Neural Computation*, Vol.2, No.4, p.447 (1990).

[Pollack 90] J.B. Pollack, The Induction of Dynamical Recognizers, Tech Report 90-JP-Automata, Dept of Computer and Information Science, Ohio State U. (1990).

[Rivest 87] R.L. Rivest, R.E. Schapire, Diversity-based Inference of Finite Automata, *Proceedings of the Twenty-Eight Annual Symposium on Foundations of Computer Science*, p. 78 (1987).

[Sun 90] G.Z. Sun, H.H. Chen, C.L. Giles, Y.C. Lee and D. Chen, Connectionist Pushdown Automata that Learn Context-Free Grammars, *Proceedings of the International Joint Conference on Neural Networks*, IJCNN-90-WASH-DC, Lawrence Erlbaum, Hillsdale, N.J., Vol I, p.577 (1990).

[Tomita 82] M. Tomita, Dynamic Construction of Finite-state Automata from Examples Using Hill-climbing. *Proceedings of the Fourth Annual Cognitive Science Conference* p.105 (1982).

[Watrous 92] R.L. Watrous and G.M. Kuhn, Induction of Finite-State Languages Using Second-Order Recurrent Networks, *Neural Computation* (1992).

[Williams 89] R.J. Williams, D. Zipser, A Learning Algorithm for Continually Running Fully Recurrent Neural Networks, *Neural Computation*, Vol.1, No.2, p.270, (1989).