Sequence Learning with Incremental Higher-Order Neural Networks

Mark Ring
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
(ring@cs.utexas.edu)

January, 1993
AI 93–193

Abstract

An incremental, higher-order, non-recurrent neural-network combines two properties found to be useful for sequence learning in neural-networks: higher-order connections and the incremental introduction of new units. The incremental, higher-order neural-network adds higher orders when needed by adding new units that dynamically modify connection weights. The new units modify the weights at the next time-step with information from the previous step. Since a theoretically unlimited number of units can be added to the network, information from the arbitrarily distant past can be brought to bear on each prediction. Temporal tasks can thereby be learned without the use of feedback, in contrast to recurrent neural-networks. Because there are no recurrent connections, training is simple and fast. Experiments have demonstrated speedups of two orders of magnitude over recurrent networks.

1 Introduction

Traditional first-order, recurrent neural-networks [2, 3, 9, 10, 12] can be improved in two notable ways. The first is by adding higher-order connections. The other is by adding units to the network incrementally as learning progresses. Second-order recurrent networks have proven to be very powerful [7]. The input to a neuron in a traditional, first-order recurrent network is the sum of first-order products, (i.e., each term in the sum is simply the product of a weight and a neuron's output).

$$\mathbf{in}^i(t+1) = \sum_j w_{ij}\mathbf{out}^j(t),$$

where $\mathbf{in}^i(t+1)$ is the input to the i^{th} neuron at the next time-step; $\mathbf{out}^j(\mathrm{t})$ is the output of the j^{th} neuron at the current time-step; and w_{ij} is the weight of the connection from unit j to unit i.

The input to a neuron of a second-order recurrent network, on the other hand, is the sum of second-order products, (i.e., each term is the product of a weight and the output of *two* units):

$$\mathbf{in}^i(t+1) = \sum_j \sum_k w_{ijk} \mathbf{out}^j(t) \mathbf{out}^k(t).$$

The second-order terms seem to have a notably positive effect on the networks, which have been shown to learn difficult tasks with a small number of training samples [1, 5, 11]. The networks are cumbersome, however, having $O(n^3)$ weights (where n is the number of neurons), and in order to get good performance, true gradient descent must be done [10, 12], which is also quite cumbersome.

A different method for getting good performance in a recurrent neural-network is to add new units during training in order to minimize error. This is done by Fahlman's Recurrent Cascade Correlation (RCC) algorithm [4]. In the RCC network, a pool of units are trained to predict the network's error, and the best is then added as a hidden unit into the network. The new unit's weights are frozen and training for the next unit begins. Each hidden unit receives input from all "input" units and all previously added hidden units. Each unit also has a single recurrent connection from itself at the previous time-step, which allows the network to learn temporal tasks.

The incremental, higher-order network presented here combines advantages of both of these approaches, but it does so in a *non-recurrent* network. This network (a simplified, continuous version of that introduced in [8]), adds higher orders when they are needed by the system to solve its task. This is done by adding new units that dynamically modify connection weights. The new units modify the weights at the next time-step with information from the last, which allows temporal tasks to be learned without the use of feedback. Long time-spans are bridged by adding the units hierarchically so that weights can be modified with information from the arbitrarily distant past.

2 General Formulation

Each unit (U) in the network is either an input (I), output (O), or high-level (L) unit.

$$\begin{aligned}
U^i(t) &\stackrel{def}{=} \text{value of the } i\text{th unit at time } t. \\
I^i(t) &\stackrel{def}{=} U^i(t) \text{ where } i \text{ is an input unit.} \\
O^i(t) &\stackrel{def}{=} U^i(t) \text{ where } i \text{ is an output unit.} \\
T^i(t) &\stackrel{def}{=} \text{Target value for } O^i(t) \text{ at time } t. \\
L^i_{xy}(t) &\stackrel{def}{=} U^i(t) \text{ where } i \text{ is the higher-order unit that} \\
&\phantom{\stackrel{def}{=}} \text{ modifies weight } w_{xy} \text{ at time } t.^1
\end{aligned}$$

[1] A connection may be modified by at most one L unit. Therefore L^i_{xy}, L^i, and L_{xy} are identical but used as appropriate for notational convenience.

The output and high-level units are collectively referred to as non-input (N) units:

$$N^i(t) \stackrel{def}{=} \begin{cases} O^i(t) & \text{if } U^i \equiv O^i. \\ L^i_{xy}(t) & \text{if } U^i \equiv L^i_{xy}. \end{cases}$$

In a given time-step, the output and high-level units receive a summed input from the input units:

$$N^i(t) = \sum_j I^j(t) g(i,j,t). \tag{1}$$

g is a gating function representing the weight of a particular connection at a particular time-step. If there is a higher-order unit assigned to that connection, then the input value of that unit is added to the connection's weight at that time-step.[2]

$$g(i,j,t) = \begin{cases} w_{ij}(t) + L^n_{ij}(t-1) & \text{If } L^n_{ij} \text{ exists} \\ w_{ij}(t) & \text{Otherwise} \end{cases} \tag{2}$$

At each time-step, the values of the output units are calculated from the input units and the weights (possibly modified by the activations of the high-level units from the previous time-step). The values of the high-level units are calculated at the same time in the same way. The output units generate the output of the network. Each high-level unit simply alters a single weight at the next time-step. All unit activations can be computed simultaneously since the activations of the L units are not required until the following time-step. The network is arranged hierarchically in that every higher-order unit is always higher in the hierarchy than the units on either side of the weight it affects. Since higher-order units have no outgoing connections, the network is not recurrent. It is therefore impossible for a high-level unit to affect, directly or indirectly, its own input.

There are no hidden units in the traditional sense, and all units have a linear activation function. (This does not imply that non-linear functions cannot be represented, since non-linearities do result from the multiplication of higher-level and input units in Equations 1 and 2.)

Learning is done through gradient descent to reduce the sum-squared error. The calculation of the learning rule is as follows. At each time-step, the weights should be changed in the direction opposite of

[2] It can be seen that this is a higher-order connection in the usual sense if one substitutes the right-hand side of Equation 1 for L^n_{ij} in Equation 2 and then replaces g in Equation 1 with the result:

$$\begin{aligned} N^i(t) &= \sum_j I^j(t)(w_{ij}(t) + \sum_{j'} I^{j'}(t-1) g(n,j',t-1)) \text{ if } L^n_{ij} \text{ exists.} \\ &= \sum_j (I^j(t) w_{ij}(t) + \sum_{j'} I^j(t) I^{j'}(t-1) g(n,j',t-1)) \text{ if } L^n_{ij} \text{ exists.} \end{aligned}$$

In fact, as the network increases in height, ever higher orders are introduced, while lower orders are preserved.

their contribution to the error, $E(t)$.

$$
\begin{aligned}
E(t) &= \frac{1}{2}\sum_i (T^i(t) - O^i(t))^2 \\
w_{ij}(t+1) &= w_{ij}(t) - \eta \Delta w_{ij}(t) && (3) \\
\Delta w_{ij}(t) &= \sum_{\tau=0}^{t} \frac{\partial E(t)}{\partial w_{ij}(\tau)} , && (4)
\end{aligned}
$$

where η is the learning rate. Since it may take several time-steps for the value of a weight to affect the network's output and therefore the error, Equation 4 can be rewritten as:

$$
\Delta w_{ij}(t) = \frac{\partial E(t)}{\partial w_{ij}(t-\tau^i)} , \tag{5}
$$

where

$$
\tau^i = \begin{cases} 0 & \text{if } U^i \equiv O^i \\ 1 + \tau^x & \text{if } U^i \equiv L^i_{xy} \end{cases} .
$$

The value τ^i is constant for any given unit i and specifies how "high" in the hierarchy unit i is. It therefore also specifies how many time-steps it takes for a change in unit i's activation to affect the network's output.

The remainder of the gradient derivation is as follows:

$$
\begin{aligned}
\frac{\partial E(t)}{\partial w_{ij}(t-\tau^i)} &= \frac{\partial E(t)}{\partial N^i(t-\tau^i)} \frac{\partial N^i(t-\tau^i)}{\partial w_{ij}(t-\tau^i)} \\
&= \delta^i(t) \frac{\partial N^i(t-\tau^i)}{\partial w_{ij}(t-\tau^i)} ,
\end{aligned}
$$

where

$$
\delta^i(t) \stackrel{def}{=} \frac{\partial E(t)}{\partial N^i(t-\tau^i)} .
$$

The second factor can be derived simply as:

$$
\begin{aligned}
\frac{\partial N^i(t-\tau^i)}{\partial w_{ij}(t-\tau^i)} &= I^j(t-\tau^i) \frac{\partial g(i,j,t-\tau^i)}{\partial w_{ij}(t-\tau^i)} \\
&= I^j(t-\tau^i) \begin{cases} 1 + \frac{\partial L^i_{xy}(t-\tau^i-1)}{\partial w_{ij}(t-\tau^i)} & \text{if } L^i_{xy} \text{ exists} \\ 1 & \text{otherwise} \end{cases} \\
&= I^j(t-\tau^i),
\end{aligned}
$$

since $t - \tau^i$ follows $t - \tau^i - 1$ in time. Therefore

$$
\Delta w_{ij}(t) = \frac{\partial E(t)}{\partial w_{ij}(t-\tau^i)} = \delta^i(t) I^j(t-\tau^i) . \tag{6}
$$

Now, $\delta^i(t)$ can be derived as follows:

$$\delta^i(t) = \begin{cases} \dfrac{\partial E(t)}{\partial O^i(t-\tau^i)} & \text{If } U^i \equiv O^i \\ \dfrac{\partial E(t)}{\partial L^i_{xy}(t-\tau^i)} & \text{If } U^i \equiv L^i_{xy} \end{cases}$$

$$\begin{aligned} \frac{\partial E(t)}{\partial O^i(t-\tau^i)} &= \frac{\partial E(t)}{\partial O^i(t)} \\ &= O^i(t) - T^i(t) \end{aligned} \tag{7}$$

$$\begin{aligned} \frac{\partial E(t)}{\partial L^i_{xy}(t-\tau^i)} &= \frac{\partial E(t)}{\partial g(x,y,t-\tau^i+1)} \frac{\partial g(x,y,t-\tau^i+1)}{\partial L^i_{xy}(t-\tau^i)} \\ &= \frac{\partial E(t)}{\partial g(x,y,t-\tau^i+1)} \\ &= \frac{\partial E(t)}{\partial N^x(t-\tau^i+1)} \frac{\partial N^x(t-\tau^i+1)}{\partial g(x,y,t-\tau^i+1)} \\ &= \frac{\partial E(t)}{\partial N^x(t-\tau^x)} \frac{\partial N^x(t-\tau^x)}{\partial g(x,y,t-\tau^x)} \\ &= \delta^x(t) I^y(t-\tau^x) \\ &= \Delta w_{xy}(t). \end{aligned} \tag{8}$$

The change to the weight can therefore be written as:

$$\begin{aligned} \Delta w_{ij}(t) &= \delta^i(t) I^j(t-\tau^i) \\ &= I^j(t-\tau^i) \begin{cases} T^i(t) - O^i(t) & \text{If } U^i \equiv O^i \\ \Delta w_{xy}(t) & \text{If } U^i \equiv L^i_{xy} \end{cases} \end{aligned} \tag{9}$$

The weights are changed after error values for the output units have been collected. Since each high-level unit is higher in the hierarchy than the units on either side of the weight it affects, weight changes are made bottom up, and the $\Delta w_{xy}(t)$ in Equation 9 will already have been calculated at the time $\Delta w_{ij}(t)$ is computed.

The intuition behind the learning rule is that each high-level unit learns to utilize the context from the previous time-step for adjusting the connection it influences at the next time-step so that it can minimize the connection's error in that context. Therefore, if the information necessary to decide the correct value of a connection at one time-step is available at the previous time-step, then that information is used by the higher-order unit assigned to that connection. If the needed information is not available at the previous time-step, then new units may be built to look for the information at still earlier steps.

3 When to Add New Units

A unit is added whenever a weight is being pulled strongly in opposite directions (i.e. when learning is forcing the weight to increase and to decrease at the same time). The unit is created to determine the contexts in which the weight is pulled in each direction. In order to decide when a new unit should be added, two long-term averages are kept for each connection. The first of these records the average change made to the weight,

$$\overline{\Delta w_{ij}(t)} = \sigma \Delta w_{ij}(t) + (1-\sigma)\overline{\Delta w_{ij}(t-1)}; \quad 0 \le \sigma \le 1.$$

The second is the long-term mean absolute deviation, given by:

$$\overline{|\Delta w_{ij}(t)|} = \sigma |\Delta w_{ij}(t)| + (1-\sigma)\overline{|\Delta w_{ij}(t-1)|}; \quad 0 \le \sigma \le 1.$$

The parameter, σ, specifies the duration of the long-term average. A lower value of σ means that the average is kept for a longer period of time. When $\overline{\Delta w_{ij}(t)}$ is small, but $\overline{|\Delta w_{ij}(t)|}$ is large, then the weight is being pulled strongly in conflicting directions, and a new unit is built.

$$\text{if} \quad \frac{\overline{|\Delta w_{ij}(t)|}}{\epsilon + |\overline{\Delta w_{ij}(t)}|} > \Theta \quad \text{then build unit } L_{ij}^{N+1}, \tag{10}$$

where ϵ is a small constant that keeps the denominator from being zero, Θ is a threshold value, and N is the number of units in the network. A related method for adding new units, but in feed-forward neural-networks, was introduced by Wynne-Jones [13].

When a new unit is added, its incoming weights are initially zero. It has no output weights but simply learns to anticipate and reduce the error at each time-step of the weight it modifies. In order to keep the number of new units low, whenever a unit, L_{ij}^{n} is created, the statistics for all connections into the destination unit (U^i) are reset: $\overline{|\Delta w_{ij}(t)|} \leftarrow 0.0$ and $\overline{\Delta w_{ij}(t)} \leftarrow \Theta$.

4 The Algorithm

Because of the simple learning rule and method of adding new units, the learning algorithm is very straightforward. The outline of the procedure is as follows:

For (Ever)
1) Get inputs.
2) Propagate Forward.
3) Get Targets.
4) Calculate Weight Changes.
5) Change Weights & Weight Statistics;
Create New Units.

The first and third of these are trivial and depend on the task being performed. The second step, forward propagation, is nearly the same as the forward propagation in standard feed-forward neural-networks:

2) Propagate Forward

```
/* First reset old O values to zero.     */
For each Output unit, O(i)
    O(i) ← zero;

/* Then calculate new output values. */
For each Non-input unit, N(i)
    For each Input unit, I(j)
        /* UnitFor(i, j) returns the input of unit L_ij at the last time-step. */
        /* Zero is returned if the unit did not exist. (See Equation 2.)       */
        L ← UnitFor(i,j);

        /* To N^i's input, add I^j's value times the (possibly modified) */
        /* weight from j to i. (See Equation 1.)                          */
        N(i) ← N(i) + I(j)*(L + Weight(i, j));
```

In order for weight changes to be done properly, each Δw_{xy} must be computed before the weights to higher-level unit L^i_{xy} are computed (because of Equation 9). Therefore, assume that the units are arranged such that the input, output and higher-level units are concatenated into a single vector; and $i < j < k,\ \forall\ I^i,\ O^j,\ L^k$. Further assume that whenever a unit is added to the network, it is appended to the end of the vector. In this case, $i < n$ and $j < n,\ \ \forall L^n_{ij}$, which means that the δ^i's and Δw_{ij}'s of Equations 6 and 9 can be computed with i in ascending order.

4) Calculate Weight Changes

```
/* Calculate δ_i for the Output units, O_i. (See Equation 7.)*/
For each Output unit, O(i)
    delta(i) = O(i) - Target(i);

/* Calculate Δw_ij s, and δ^i's for higher-order units L_i.    */
For each Non-input unit, N(i), with i in ascending order
    For each Input unit, I(j)
        /* Compute weight change (Equation 6). Previous(j, i) retrieves I^j(t − τ^i). */
        delta_w(i, j) ← delta(i) * Previous(j, i);

        /* Compute δ^n for L^n_ij if it exists. IndexOfUnitFor(i, j) returns n; or zero if L^n_ij does not exist. */
        n ← IndexOfUnitFor(i, j);
        if n > 0
            delta(n) ← delta_w(i, j);
```

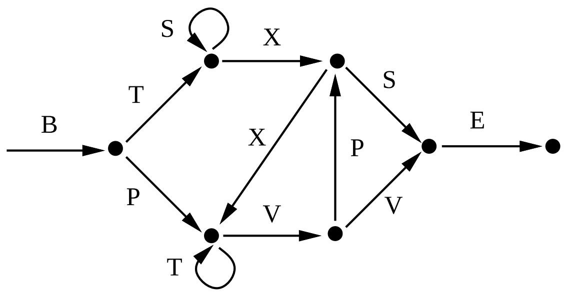
Finally, the weight changes need to be made. Weight statistics for each weight are updated at the same time the weight is changed. The statistics are also checked to see if a new unit is warranted. If a unit *is* warranted for the weight leading from unit j to unit i, a unit is built for it, and the statistics are reset for all weights leading into unit i.

5) Change Weights & Weight Statistics; Create New Units.

```
For each Non-input unit, N(i)
    For each Input unit, I(j)

        /* Change weight w_ij. (See Equation 3.) */
        Weight(i, j) ← Weight(i, j) - LEARNING_RATE * delta_w(i, j);

        /* Update ‾Δw_ij‾, the long-term average of Δw_ij. */
        lta(i, j) ← SIGMA * delta_w(i, j) + (1 - SIGMA) * lta(i, j);

        /* Update ‾|Δw_ij|‾, the long-term mean absolute deviation. */
        ltmad(i, j) ← SIGMA * abs(delta_w(i, j)) + (1 - SIGMA) * ltmad(i, j);

        /* If Higher-Order unit L^n_ij does not exist ... */
        if IndexOfUnitFor(i, j) = 0
            /* ... but such a unit should exist (Equation 10) ... */
            if (ltmad(i, j) / (EPSILON + abs(lta(i, j)))) > THRESHOLD
                /* ... then create unit L^{N+1}_ij, (where N is the number of units in the network).*/
                BuildUnitFor(i, j);

                /* Reset statistics for all incoming weights. */
                For each input unit, I(k)
                    lta(i, k) ← THRESHOLD;
                    ltmad(i, k) ← 0.0;
```

5 Results

The Reber grammar is a small finite-state grammar of the following form:



Transitions from one node to the next are made by way of the labeled arcs. The task of the network is: given as input the label of the arc just traversed, predict the arc that will be traversed next. A training sequence,

Algorithm:	Elman Network	RTRL	Recurrent Cascade Correlation	Incremental Higher-Order Network
Mean Number of Training Strings:	–	–	25,000	206
Best Number of Training Strings:	20,000	19,000	–	176
Hidden or High-Level Units:	15	2	2-3	40

Table 1: The incremental higher-order network is compared against recurrent networks on the Reber grammar. The results for the recurrent networks are quoted from other sources [2, 4]. The mean and/or best performance is shown when available. RTRL is the real-time recurrent learning algorithm [12].

or *string,* is generated by starting with a B transition and then randomly choosing an arc leading away from the current state until the final state is reached. Both inputs and outputs are encoded locally, so that there are seven output units (one each for B, T, S, X, V, P, and E) and eight input units (the same seven plus one bias unit). The network is considered correct if its highest activated outputs correspond to the arcs that can be traversed from the current state. Note that the current state cannot be determined from the current input alone.

An Elman-type recurrent network was able to learn this task after 20,000 string presentations using 15 hidden units [2]. (The correctness criteria for the Elman net was slightly more stringent than that described in the previous paragraph.) Recurrent Cascade-Correlation (RCC) was able to learn this task using only two or three hidden units in an average of 25,000 string presentations [4].

The incremental, higher-order network was trained on a continuous stream of input: the network was not reset before beginning a new string. Training was considered to be complete only after the network had correctly classified 100 strings in a row. Using this criterion, the network completed training after an average of 206.3 string presentations with a standard deviation of 16.7. It achieved perfect generalization on test sets of 128 randomly generated strings in all ten runs. Because the Reber grammar is stochastic, a ceiling of 40 higher-order units was imposed on the network to prevent it from continually creating new units in an attempt to outguess the random number generator.

Complete results for the network on the Reber grammar task are given in Table 1. The parameter settings were: $\eta = 0.04, \sigma = 0.08, \Theta = 1.0, \epsilon = 0.1$ and Bias = 0.0. (The network seemed to perform better with no bias unit.)

The network has also been tested on the "variable gap" tasks introduced by Mozer [6], as shown in Figure 1. These tasks were intended to test performance of networks over long time-delays. Two sequences are alternately presented to the network. Each sequence begins with an X or a Y and is followed by a fixed string of characters with an X or a Y inserted some number of time-steps from the beginning. In Figure 1 the number of time-steps, or "gap", is 2. The only difference between the two sequences is that the first begins with an X and repeats the X after the gap, while the second begins with a Y and repeats the Y after the gap. The network must learn to predict the next item in the sequence given the current item as input (where all inputs are locally encoded). In order for the network to predict the second occurrence of the X

Time-step:	0	1	2	3	4	5	6	7	8	9	10	11	12
Sequence 1:	X	a	b	X	c	d	e	f	g	h	i	j	k
Sequence 2:	Y	a	b	Y	c	d	e	f	g	h	i	j	k

Figure 1: An example of a "variable gap" training sequence [6]. One item is presented to the network at each time-step. The target is the next item in the sequence. Here the "gap" is two, because there are two items in the sequence between the first X or Y and the second X or Y. In order to correctly predict the second X or Y, the network must remember how the sequence began.

or Y, it must remember how the sequence began. The length of the gap can be increased in order to create tasks of greater difficulty.

Results of the "gap" tasks are given in Table 2. The values for the standard recurrent network and for Mozer's own variation are quoted from Mozer's paper [6]. Mozer reported results for gaps up to ten, and he also stated that the network had scaled linearly even up to gaps of 25. The incremental higher-order net was tested on gaps up to 24. The training string was composed of the letters of the alphabet in order. The same string was used for all tasks (except for the position of the second X or Y), and had no repeated characters (again with the exception of the X and Y). For example, with a gap of twelve, the strings looked like this:

`XabcdefghijklXmnopqrztuvwz` and `YabcdefghijklYmnopqrztuvwz.`

The network continued to scale linearly with every gap size both in terms of units and epochs required for training. Because these tasks are not stochastic, the network always stopped building units as soon as it had created those necessary to solve each task.

Mozer used different parameter settings for each gap size and reported that the network required precise tuning for the largest gaps. The parameter settings for the incremental higher-order network, which were constant across all gap sizes, were: $\eta = 1.5, \sigma = 0.2, \Theta = 1.0, \epsilon = 0.1$ and Bias $= 0.0$. The network

Gap	Mean number of Training sets required by: Standard Recurrent Net	Mozer Net	Incremental Higher-Order Net	Units Created by Incremental Higher-Order Net
2	468	328	4	10
4	7406	584	6	15
6	9830	992	8	19
8	> 10000	1312	10	23
10	> 10000	1630	12	27
…			…	…
24			26	49

Table 2: The incremental higher-order network is compared on the "variable gap" task against a standard recurrent network [10] and a network devised specifically for learning long time-delays [6]. The comparison values are quoted from Mozer [6], who reported results for gaps up to ten.

was considered to have correctly predicted an element in the sequence if the most strongly activated output unit was the unit corresponding to the correct prediction. The sequence was considered correctly predicted if all elements (other than the initial X or Y) were correctly predicted.

6 Conclusions

The incremental higher-order network performed much better than the networks that it was compared against on these tiny tests. A few caveats are in order, however. First, the parameters given above were optimized for these tasks, and they demonstrate the wide-range of settings that may be needed for different kinds of tasks. Second, the network may add a large number of new units if the task contains many context-dependent events, or if it is inherently stochastic. Third, though the network in principle can build an ever larger hierarchy that searches further and further back in time for a context that will predict what a connection's weight should be, many units may be needed to bridge a long time-gap. Finally, once a bridge across a time-delay is created, it does not generalize to other time-delays. This means that the network would have great difficulty learning the embedded Reber grammar [2, 4], where a variable number of time-steps can occur between the presentation of important information and the moment that the information is eventually used. These four issues are the subject of current research.

On the other hand, the network learns very fast due to its simple structure that adds high-level units only when needed. Since there is no feedback (i.e. no unit ever produces a signal that will ever feed back to itself), learning can be done without back propagation through time. Also, since the outputs and high-level units have a fan-in equal to the number of inputs only, the number of connections in the system is much smaller than the number of connections in a traditional network with the same number of hidden units.

Finally, the network can be thought of as a system of continuous-valued condition-action rules that are inserted or removed depending on another set of such rules that are in turn inserted or removed depending on another set, etc. When new rules (new units) are added, they are initially invisible to the system, (i.e., they have no effect), but only gradually learn to have an effect as the opportunity to decrease error presents itself.

Acknowledgements

This work was supported by NASA Johnson Space Center Graduate Student Researchers Program training grant, NGT 50594. I would like to thank Eric Hartman, Kadir Liano, and my advisor Robert Simmons for useful discussions and helpful comments on drafts of this paper. I would also like to thank Pavilion Technologies, Inc. for their generous contribution of computer time and office space required to complete much of this work.

References

[1] Jonathan Richard Bachrach. *Connectionist Modeling and Control of Finite State Environments.* PhD thesis, Department of Computer and Information Sciences, University of Massachusetts, February 1992.

[2] Axel Cleeremans, David Servan-Schreiber, and James L. McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381, 1989.

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

[4] Scott E. Fahlman. The recurrent cascade-correlation architecture. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 190–196, San Mateo, California, 1991. Morgan Kaufmann Publishers.

[5] C. L. Giles, C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen, and Y. C. Lee. Extracting and learning an unknown grammar with recurrent neural networks. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 317–324, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[6] Michael C. Mozer. Induction of multiscale temporal structure. In John E. Moody, Steven J. Hanson, and Richard P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 275–282, San Mateo, California, 1992. Morgan Kaufmann Publishers.

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

[8] Mark B. Ring. Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. In Lawrence A. Birnbaum and Gregg C. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pages 343–347. Morgan Kaufmann Publishers, June 1991.

[9] 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.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. V1: Foundations.* MIT Press, 1986.

[11] Raymond L. Watrous and Gary M. Kuhn. Induction of finite-state languages 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, California, 1992. Morgan Kaufmann Publishers.

[12] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[13] Mike Wynn-Jones. Node splitting: A constructive algorithm for feed-forward neural networks. *Neural Computing and Applications*, 1(1):17–22, 1993.