Experimental Comparison of the Effect of Order in Recurrent Neural Networks

Clifford B. Miller[a] and C.L. Giles[a,b]
[a]NEC Research Institute, 4 Independence Way, Princeton, NJ 08540
[b]Institute for Advanced Computer Studies, U. Maryland, College Park, MD20742

Abstract

There has been much interest in increasing the computational power of neural networks. In addition there has been much interest in "designing" neural networks to better suit particular problems. Increasing the "order" of the connectivity of a neural network permits both. Though order has played a significant role in feedforward neural networks, it role in dynamically driven recurrent networks is still being understood. This work explores the effect of order in learning grammars. We present an experimental comparison of first-order and second-order recurrent neural networks, as applied to the task of grammatical inference. We show that for the small grammars studied these two neural net architectures have comparable learning and generalization power, and that both are reasonably capable of extracting the correct finite state automata for the language in question. However, for a larger randomly-generated, ten-state grammar second-order networks significantly outperformed the first-order networks, both in convergence time and generalization capability. We show that these networks learn faster the more neurons they have (our experiments used up to 10 hidden neurons), but that the solutions found by smaller networks are usually of better quality (in terms of generalization performance after training). Second-order nets have the advantage that they converge more quickly to a solution and can find it more reliably than first-order nets, but that the second-order solutions tend to be of poorer quality than those of first-order if both architectures are trained to the same error tolerance. Despite this, second-order nets can more successfully extract finite state machines using heuristic clustering techniques applied to the internal state representations. We speculate that this may be due to restrictions on the ability of first-order architecture to fully make use of its internal state representation power and that this may have implications for the performance of the two architectures when scaled up to larger problems.

List of key words:

recurrent neural networks, higher order, learning, generalization, automata, grammatical inference, grammars.

Published in:
"Special Issue on Applications of Neural Networks to Pattern Recognition", *International Journal of Pattern Recognition and Artificial Intelligence*, vol 7., no. 4. p. 849, 1993.

Experimental Comparison of the Effect of Order in Recurrent Neural Networks

Clifford B. Miller
cliff@research.nj.nec.com

C. Lee Giles*
giles@research.nj.nec.com

NEC Research Institute, 4 Independence Way, Princeton, N.J. 08540 USA
*Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland

Abstract

There has been much interest in increasing the computational power of neural networks. In addition there has been much interest in "designing" neural networks to better suit particular problems. Increasing the "order" of the connectivity of a neural network permits both. Though order has played a significant role in feedforward neural networks, it role in dynamically driven recurrent networks is still being understood. This work explores the effect of order in learning grammars. We present an experimental comparison of first-order and second-order recurrent neural networks, as applied to the task of grammatical inference. We show that for the small grammars studied these two neural net architectures have comparable learning and generalization power, and that both are reasonably capable of extracting the correct finite state automata for the language in question. However, for a larger randomly-generated, ten-state grammar second-order networks significantly outperformed the first-order networks, both in convergence time and generalization capability. We show that these networks learn faster the more neurons they have (our experiments used up to 10 hidden neurons), but that the solutions found by smaller networks are usually of better quality (in terms of generalization performance after training). Second-order nets have the advantage that they converge more quickly to a solution and can find it more reliably than first-order nets, but that the second-order solutions tend to be of poorer quality than those of first-order if both architectures are trained to the same error tolerance. Despite this, second-order nets can more successfully extract finite state machines using heuristic clustering techniques applied to the internal state representations. We speculate that this may be due to restrictions on the ability of first-order architecture to fully make use of its internal state representation power and that this may have implications for the performance of the two architectures when scaled up to larger problems.

Keywords: recurrent neural networks, higher order, learning, generalization, automata, grammatical inference, grammars.

1 Introduction

Because of their innate ability to model "internal state" information and process temporal signals, there has been much recent interest in recurrent neural network models. Although they operate only in discrete time, such networks are naturally dynamic and thus readily extend the computational power of feed-forward neural networks. Interestingly, one of the first neural network models was recurrent (McCulloch and Pitts, 1943, [33]). The representational issues of recurrent neural networks to finite state automata was further explored by Kleene [27] and Minsky [35], and recently extended to cellular automata [10, 18]. Then there

was much initial interest in steady-state recurrent network models [24] for optimization problems. Since then there has been much work on how recurrent neural networks are related to and learn automata; see for example [6, 9, 17, 15, 25, 26, 32, 37, 45, 51, 53].

The focus of this paper is on the *order* of the recurrent neural network models and how well this type of order relates to the performance of recurrent neural network in learning temporal sequences. There could be many definitions of order in neural network models. Our definition of order is the same as that of Minsky & Papert [36], and the polynomial order of Cover [7]. (A similar but more restrictive definition is that of "sigma-pi" units [49].) In general these higher order extensions have enhanced the computational power and performance of feed-forward neural networks but usually at the cost of lessening the generality of the net and increased net complexity (for some examples see [14, 48]). This introduction of order and its selected use can be interpreted as method for "designing" a network to suit a specific problem. These feedforward "higher order" neural networks have found successful use in many models and applications, from placing *a priori* knowledge into a neural network [21, 28, 39, 41, 42] to modeling synaptic interconnections [4, 28, 43].

It is straightforward to extend this definition of order to recurrent neural network models [5, 44, 20, 31, 48]. Until recently, there was little evidence that increasing the order in a dynamic recurrent network (associative memory is another issue) gave any of the computational advantages seen in higher-order feed-forward networks. Representational and experimental arguments could be made for using higher order recurrent networks to learn automata [8, 17, 15, 46, 45, 50, 51, 53]. In particular, Goudreau *et al* [19] show that for hard-threshold neurons, a single-layer first-order recurrent neural network cannot represent all finite state automata. In addition, recent experimental comparisons of grammar learning [8] showed that higher order made it easier to learn context-free grammars using a neural network pushdown automaton. The motivation of this work was to give an experimental comparison of order in simple recurrent neural networks.

In order to compare learning performance of the higher-order architectures, we chose the testbed problem of grammatical inference, in which the goal is to infer a grammar with an *inference engine*, given a sample of strings generated by the grammar. If the sample contains both positive and negative examples, then in the worst case the problem is NP-hard. See [2, 13, 34] for a discussion of difficulty and existing methods of solution. (Also see [30] for a very promising new method and impressive results.) For a discussion of grammars, finite state automata and languages, see a text such as [23]. In order to determine how well the grammar has been learned, one can either extract the grammar from the inference engine which learned the

grammar, or test how well the inference engine classifies previously unseen strings – the task of generalization. In this study we do both – perform generalization and extract the grammar (in the form of a finite state automaton) from the neural network. For our purposes, the inference engine is a recurrent neural network of first or second order (defined below).

2 Dynamic Recurrent Neural Networks

2.1 Definition of Recurrent Neural Networks

A dynamic recurrent neural network (RNN) can processes temporal input sequences of arbitrary length that vary over (discrete) time. (Steady-state recurrent networks cannot handle sequences of arbitrary length.) See [22] for a discussion of dynamic recurrent network models. A recurrent neural network consists of a set of N recurrent "state" neurons, some serving as "output" units and the rest as "hidden" units, and a set of L *non*-recurrent "input" neurons which receive signals from outside the network. Both hidden and output units feed back into themselves through a set of synchronous delay lines (see Figure 1). Thus, our model of a RNN can be thought of as a feedforward network with an arbitrary (potentially infinite) number of identical layers; the weight W_{ij} of any given neural interconnect $i \rightarrow j$ has the same value on each layer.

In this simple fully-connected RNN, all state neurons are functionally equivalent and symmetrically connected; the only real distinction between output and hidden units is at time steps when certain neurons are singled out to show "target" values. This distinction is conceptual only – in other words, *all* the state neurons always feed into the next layer (time step) with full connectivity, regardless of the designation of output neurons. One could also create additional *nonrecurrent* units to act as outputs on different time steps. These "output time steps" could occur at any time during a temporal input sequence – they are defined by the problem at hand. In classification problems, output usually occurs only at the end of the sequence, or at periodic intervals. In transduction (translation) problems, one might expect outputs at most (or possibly all) time steps. We present an experimental performance comparison of the "first-order" and "second-order" RNN architectures in a classification problem. (See [19, 50] for some theoretical comparisons.)

2.2 The Definition of "Order"

The "order" of a neural network refers to the dimensionality of product terms in the weighted sum, which reflects the connectivity of the network. Note that all discussions of RNN's in this paper are restricted to fully-connected RNN's; hence the connectivity is the same throughout the network at all time steps. The

recurrence equation for a RNN, which defines its temporal dynamics, typically filters a weighted sum of states and/or inputs through a nonlinear "discriminant" function. This function has the general form:

$$\mathbf{S}^{(t+1)} = \mathbf{F}\left(\mathbf{S}^{(t)}, \mathbf{I}^{(t)}; \mathbf{W}, \mathbf{\Theta}\right) \tag{1}$$

where $\mathbf{S}^{(t)} \in \Re^N$ represents the values of all state neurons (the "state vector") at time t, $\mathbf{I}^{(t)} \in \Re^L$ the values of all input neurons (the "input vector") at time t, $\mathbf{F}$ is a vector function (usually a nonlinear mapping), $\mathbf{W}$ is a set of weight matrices defining the weighted interconnects between layers, and $\mathbf{\Theta}$ is a set of biases on the state neurons. It is important to note that Equation 1 above is the standard definition of the {state;input $\rightarrow$ next-state} mapping found in the definitions of both finite state automata [29] and nonlinear systems models [38]. In addition, for each type of recurrent neural network we discuss, there exists *only one hidden layer* per time step. We do this in order to explore as simple a model as possible. However, a final "output" layer, with its own set of weights, is added to the network by the input encoding (discussed below), which appends an "end symbol" to an input sequence to designate the end of an input string. This gives additional degrees of freedom to the decision making process and gets around the representation limitations imposed by the first order, single hidden layer model [19].

Inspection of the recurrence equation for each type of architecture reveals their differences. For first-order recurrent nets, we define the recurrence as

$$S_i^{(t+1)} = f(Y_i^t), \qquad Y_i^t = \sum_j^N W_{ij} S_j^{(t)} + \sum_k^L V_{ik} I_k^{(t)} + \Theta_i \tag{2}$$

In this architecture, there are two independent sets of weights, one for the state neurons ($\mathbf{W}$, size $N \times N$), and one for the input neurons ($\mathbf{V}$, size $N \times L$). There is also a set of N biases $\mathbf{\Theta}$, one for each state neuron. Here f is a nonlinear *discriminant* function; in our implementation of the RNN, this function is a sigmoid, $f(x) = (1 + \exp(-x))^{-1}$, which limits the activation range of all state neurons to $0 < S_i^{(t)} < 1$.

For second order nets, our definition of the recurrence equation is

$$S_i^{(t+1)} = f(Y_i^t), \qquad Y_i^t = \sum_j^N \sum_k^L W_{ijk} S_j^{(t)} I_k^{(t)} + \Theta_i \tag{3}$$

For this type of net there is only a single set of weights $\mathbf{W}$, but its structure is more complex than either of the two sets of weights in the first-order net, since it is in fact a "higher-order" $N \times N \times L$ matrix. f and Θ_i are the same here as for the first-order case.

The first-order weight matrices have dimensionality $D(\mathbf{W}) = N^2$ and $D(\mathbf{V}) = NL$ elements for a total

of $N^2 + NL = N(N+L)$ elements. The second-order weight matrix has $D(\mathbf{W}) = N^2 L$ elements. Hence in the limit of large N, second-order RNN's have L times as many weights as first-order RNN's.

2.3 Input Encoding

A *string* is defined as a series of symbols $\sigma(t)$ presented to the neural network one per time step t. The symbols are taken from a finite discrete alphabet Σ. These symbols are transformed into neuron input vectors $\sigma(t) \rightarrow \mathbf{I}^{(t)}$ via a one-to-one mapping called an "input encoding scheme".

All results obtained in this study use "local" or "unary" input encoding (also referred to as "one-hot" encoding in VLSI [3]). In this input encoding scheme, the mapping from input symbols to input-neuron activations is a direct, one-to-one mapping: when symbol σ appears in the string, input neuron σ is turned on and all others are turned off. Symbolically this can be represented as $I_k^{(t)} = \delta_{k\sigma(t)}$ (δ signifies Kronecker delta).

This kind of encoding has important consequences in the recurrence equations of the two types of RNN's studied here. If we substitute in the Kronecker delta equation for the input activations, the recurrence equations become

$$\begin{aligned} S_i^{(t+1)} &= f\left(\sum_j^N W_{ij} S_j^{(t)} + V_{i\sigma(t)} + \Theta_i\right) \quad \text{[first-order]} \\ S_i^{(t+1)} &= f\left(\sum_j^N W_{ij\sigma(t)} S_j^{(t)} + \Theta_i\right) \quad \text{[second-order]} \end{aligned} \tag{4}$$

Note that these equations can also be expressed in vector form (where dot products take the place of the sum over j):

$$\begin{aligned} S_i^{(t+1)} &= f\left(\mathbf{W}_i \cdot \mathbf{S}^{(t)} + V_{i\sigma(t)} + \Theta_i\right) \quad \text{[first-order]} \\ S_i^{(t+1)} &= f\left(\mathbf{W}_{i\sigma(t)} \cdot \mathbf{S}^{(t)} + \Theta_i\right) \quad \text{[second-order]} \end{aligned} \tag{5}$$

In Eqns. (5), letters in boldface denote N-vectors.

These forms of the equations suggest that, when a local input encoding is used, the input has the effect of an additional, input-dependent bias in first-order RNN's, whereas in the second-order case the input acts as a selector for different banks of weights. These second-order weights enable a direct representation of the ordered triple of {*state, input, next-state*} that is fundamental in a discrete-time input state equation. They also make it very easy to directly encode any available *a priori* knowledge about state information [16] into the network. With a first order network, such encoding requires the solution of a linear equation [11, 12].

2.4 Example

We show an example of a second order recurrent neural network processing a string. The table below shows the values of input and state neurons for each time step, while processing the string "01011". The RNN uses three input neurons (two for symbols '0' and '1' and one for the end symbol) and three state neurons.

Network Weight Values

$j \longrightarrow$

$i \downarrow$

$k=0$		
5.6782	−5.0887	−1.9161
0.82318	2.8405	−0.29577
−0.90778	2.0358	−0.38172

$k=1$		
4.8922	−1.6205	−2.6996
−3.4412	−0.66346	0.53954
−2.0606	1.4843	0.76427

$k=2$		
5.2501	−0.70022	−2.8328
0.44247	0.95956	−0.60462
−0.34986	0.64068	−0.026902

State Values Computed for String "01011"

t	S_0	S_1	S_2	input	I_0	I_1	I_2
0	1.000	0.000	0.000	0	1.0	0.0	0.0
1	0.997	0.695	0.287	1	0.0	1.0	0.0
2	0.951	0.023	0.310	0	1.0	0.0	0.0
3	0.991	0.681	0.282	1	0.0	1.0	0.0
4	0.952	0.024	0.307	1	0.0	1.0	0.0
5	0.978	0.042	0.156	e	0.0	0.0	1.0
6	**0.991**	0.594	0.421	–	–	–	–

For this string, $f = 6$ is the final time step, and the value of the output neuron is $S_0^{(6)} = 0.991$.

2.5 Training a Recurrent Neural Network

In recent years, a great variety of neural-network training algorithms have been developed (for an overview see [22, 41]). Most of these algorithms apply to feedforward networks, but simple modifications extend them to recurrent networks. The training algorithm discussed here is a feedforward version of real-time, recurrent learning (RTRL) [38, 55]. We present a brief exposition of recurrent neural network training, followed by a description of our training algorithm.

From a mathematical perspective, a neural net can be thought of as a vector in high dimensional space. In the previous section we reviewed the dimensionality of the recurrent neural networks discussed in this paper. Although the threshold function $f(x)$ is a fixed entity in the neural network, the *parameters* of the function, *i.e.* the weights and biases, are adjustable. These parameters, taken together, form a vector in this high-dimensional space. The process of training a neural network is actually a search for a minimum of an energy function $E(\mathbf{W})$ defined over the parameter space. (Note that we use unsubscripted $\mathbf{W}$ to refer to

both weights *and* biases.) The "energy" in the case of language recognition is the average classification error made by the system. Later in this section we present the mathematical form of the energy function and a definition of its gradient in parameter space.

Our goal in error minimization is to reduce the error E on every training examples to $E < \epsilon$, where ϵ is some small positive real called the *error tolerance*. In some applications one might only require that the average error be minimized, but in our case since we use real-time, on-line training, we require a definite correct answer for every input string, *i.e.*, $E < \epsilon$ for *every* training example.

In our training algorithm, we determine the network's response to each training example, and if the error exceeds the tolerance ϵ, compute a *weight update*, that is, a correction to the weights and biases that reduces the network's response error. (It is important to compare this to other methods such as batch training [53] or training at each time step, as in a prediction problem [6]). We designate the output neuron as S_0, and its value at the end of the input string as $S_0^{(f)}$ (f stands for the "final" time step). Our performance criterion is then $E = |T - S_0^{(f)}| < \epsilon$, where T is the *target* value for the input string, *i.e.*, 1 for "yes" and 0 for "no".

Since the network's response varies as we change the parameters (weights and biases), the error also varies. This is the justification for viewing error as a function defined over parameter space, $E(\mathbf{W})$. The task of minimizing the error reduces to finding a global minimum of this function in parameter space. We employ a variant of gradient descent to accomplish this task. Although the gradient $\nabla_{\mathrm{w}} E$ may be expressed in closed form (as derived below), in practice we calculate it numerically for each input string based on the above recurrence equations.

Consider first using a Newton's method algorithm for descending the error surface to a minimum. This is accomplished by following the negative gradient in small increments:

$$\Delta \mathbf{W} = -\alpha \nabla_{\mathrm{w}} E \tag{6}$$

Here, α is a positive constant called the "learning rate", an experimentally adjustable parameter which defines the step size taken during gradient descent. Small values of α give slow but stable convergence on minima in parameter space; larger values may speed convergence but may also lead to instability or may overshoot small minima basins.

It turns out, however, that $E_{tot}(\mathbf{W}; \mathbf{X})$, the sum total of all individual error responses over an entire training set $\mathbf{X}$, is not in general well-behaved. This error surface is full of local minima, where the error response is very small for some input strings, but not all. During gradient descent, the trajectory of the

weight vector tends to be deflected into these local minima, and if the minimum has a large attraction basin the algorithm may not be able to escape. Large attractor basins which do not contain the global minimum of E_{tot} represent networks that perform well on a majority of strings but give a very large error for some others – clearly not optimal performance.

The problem of local minima can be overcome somewhat by using "momentum" [22], which incorporates previous gradients into the current gradient:

$$\Delta^\tau \mathbf{W} = -\alpha \nabla_{\mathrm{w}} E + \eta \Delta^{\tau-1} \mathbf{W} \tag{7}$$

We use τ as a counter of "number of weight updates computed". Here, $\Delta^{\tau-1}\mathbf{W}$ signifies the weight update $\Delta\mathbf{W}$ that was computed the last time an error was made; $\Delta^\tau \mathbf{W}$ is the current update. The "momentum factor", $0 < \eta < 1$, specifies the strength with which the previous update influences the current one. The net effect of the momentum is that the local gradient deflects the trajectory through parameter space, but does not completely dominate it.

The recursive definition of $\Delta^\tau \mathbf{W}$ can be expanded in τ to give

$$\Delta^\tau \mathbf{W} = -\alpha \sum_{k=0}^{\tau_0} \eta^k \, \nabla_{\mathrm{w}} E(\mathbf{W}^{\tau-k}) \tag{8}$$

where τ_0 is the number of weight updates made so far during training. This makes it clear that the weight update at any time τ is actually influenced by many local gradients, each evaluated at earlier points $\mathbf{W}^{\tau-k}$ along the trajectory. The result is an average or "effective" gradient with many fewer attractor basins, making it much easier to find the global minimum basin. The larger the momentum, the greater the averaging effect.

In our training algorithm, we actually minimize the squared error, $E^2 = \frac{1}{2}(T - S_0^{(f)})^2$, over all training examples. This avoids the derivative discontinuity in the absolute value $|T - S_0^{(f)}|$ and also helps to focus on larger errors in the training set. Hence the gradient becomes

$$\nabla_{\mathrm{w}} E^2 = 2E \nabla_{\mathrm{w}} E \tag{9}$$

$$= -(T - S_0^{(f)}) \nabla_{\mathrm{w}} S_0^{(f)} \tag{10}$$

Since the target value T does not depend on the weights, $\nabla_{\mathrm{w}} T = 0$.

The gradient $\nabla_{\mathrm{w}} S_0^{(f)}$ is the term we compute from the recurrence equations. The gradient vector has as many components as the weight vector, and each component is a partial derivative with respect to a single

weight. Hence we can compute these components explicitly:

$$\text{first-order} \begin{cases} \dfrac{\partial S_i^{(t+1)}}{\partial W_{lm}} = f'(Y_i^{(t)}) \cdot \left(\delta_{il} S_m^{(t)} + \displaystyle\sum_j^N W_{ij} \frac{\partial S_j^{(t)}}{\partial W_{lm}} \right) \\ \dfrac{\partial S_i^{(t+1)}}{\partial V_{ln}} = f'(Y_i^{(t)}) \cdot \left(\delta_{il} I_n^{(t)} + \displaystyle\sum_j^N W_{ij} \frac{\partial S_j^{(t)}}{\partial V_{ln}} \right) \\ \dfrac{\partial S_i^{(t+1)}}{\partial \Theta_l} = f'(Y_i^{(t)}) \cdot \left(\delta_{il} + \displaystyle\sum_j^N W_{ij} \frac{\partial S_j^{(t)}}{\partial \Theta_l} \right) \end{cases} \tag{11}$$

$$\text{second-order} \begin{cases} \dfrac{\partial S_i^{(t+1)}}{\partial W_{lmn}} = f'(Y_i^{(t)}) \cdot \left(\delta_{il} S_m^{(t)} I_n^{(t)} + \displaystyle\sum_{j,k}^{N,L} W_{ijk} \frac{\partial S_j^{(t)}}{\partial W_{lmn}} I_k^{(t)} \right) \\ \dfrac{\partial S_i^{(t+1)}}{\partial \Theta_l} = f'(Y_i^{(t)}) \cdot \left(\delta_{il} + \displaystyle\sum_{j,k}^{N,L} W_{ijk} \frac{\partial S_j^{(t)}}{\partial \Theta_l} I_k^{(t)} \right) \end{cases} \tag{12}$$

Here f' is the derivative of f, and $Y_i^{(t)}$ represents the argument of f in the original recurrence equations. We will refer to these derivatives collectively with the symbol $\nabla_{\mathrm{w}} \mathbf{S}^{(t)}$. Note that $\nabla_{\mathrm{w}} \mathbf{S}^{(0)} = \mathbf{0}$, since we assume that the initial state vector $\mathbf{S}^{(0)}$ is not dependent on the weights. (To make the initial values weight dependent would assume some *a priori* knowledge about the problem to be solved.) All subsequent terms $\nabla_{\mathrm{w}} \mathbf{S}^{(t)}$ can be computed *in parallel* with $\mathbf{S}^{(t)}$.

2.6 Training Algorithm

We present here an outline of the training algorithm used in our experiments. We take the approach that the neural network can learn more effectively if the problem is learned piece by piece rather than wholesale. So far comparisons with other results [53] seem to bear this out. Thus, we divide up the learning process into several periods called "cycles". During each cycle we train on a small subset of the training set called the "working set". Typically the working set initially contains $X_i < 100$ examples. The goal during a learning cycle is to learn all the examples in the working set with accuracy ϵ, or in other words so that $\forall x, E(x) < \epsilon$. The neural net is given a finite number of epochs to learn the working set. The cycle ends when either the working set is learned or the time limit is reached. Then the neural net is tested on the whole training set (weights frozen during testing), and if $E(x) > \epsilon$ for some examples in this set, a small number of these examples (up to X_r, the recycle-set size) are added to the working set, and a new cycle begins. Cycling continues until the entire training set is learned to within tolerance, or until the upper limit on cycles is reached.

During an epoch, the neural net may not get to train on all strings if it makes too many errors. We define

low and high error *thresholds*, E_l and E_h, and count the number of times these thresholds are exceeded. If at least five large errors (*i.e.* $E > E_h$) and 30 small errors ($E > E_l$), the epoch is then terminated and the neural net instructed to begin again at the beginning of the working set. This allows the size of the working set to actually contract temporarily during learning. This often helps the network to avoid forgetting examples it has already learned as the working set grows.

A pseudocode outline of the training algorithm is presented below.

```
select initial working set
select initial weights

repeat up to MAXCYCLES cycles {
/* training phase */
       repeat up to MAXEPOCHS epochs {
              large_err_count=small_err_count=0;
              for n=1 to working_set_size {
                     present string(n);
                     evaluate error E;
                     if(E>epsilon) {
                            compute gradient and weight update;
                            change weights;
                            increment small_err_count;
                            if(E>E_l) increment large_err_count;
                     }
                     if(large_err_count>=5 and small_err_count>=30)
                            break out of for loop; /* end epoch */
              }
              if(large_err_count==0 and small_err_count==0)
                     break out of epoch loop; /* end this cycle of training */
       }
/* testing phase */
       test_err_count=0;
       for n=1 to training_set_size {
              present string(n);
              evaluate error;
              if(E>epsilon) increment test_err_count;
       }
       if(test_err_count==0)
              break out of cycle loop; /* training is finished */
       else
              add error strings to working set (up to X_r)
}
```

3 Training Recurrent Neural Networks to Infer Grammars

Grammatical inference [1, 2, 13] is the problem of discovering the underlying grammar which best describes

a finite set of positive and negative example strings. A *grammar* is a set of rules for generating strings; its corresponding *language* is the complete set of strings that can be generated by that grammar. *Positive* examples are strings which belong to the language; *negative* examples are strings from the *complement language*, *i.e.* the set of all strings which *cannot* be generated by the grammar. We use both positive and negative examples to solve the inference problem.

The model and algorithms used to solve the problem of grammatical inference are collectively called the *inference engine*; we utilize first- and second-order recurrent neural networks as the model, and real-time recurrent learning as the algorithm, for the inference engine in this study.

An inference engine has successfully inferred a grammar when it not only classifies all examples from its training set correctly, but also generalizes to correctly classify any example not included in the training set. One might argue, however, that generalization can never be completely proven, since languages generally contain an infinite number of strings. A more rigorous proof of successful inference is if the engine can directly extract the grammatical rules for generating strings, which are necessarily finite. We use the method of DFA extraction from recurrent neural networks to test inference in this way.

3.1 Benchmark Grammars

In order to make comparisons of the performance characteristics of different order networks, we have selected a benchmark set of problems. Recently, a set of seven small languages introduced by Tomita [52] have become of a benchmark for recurrent neural network learning (see for example [25, 45, 53]). These seven relatively simple languages belong to the class of regular languages, whose generating grammars can be specified as discrete finite state automata (DFA). In this study we have used these languages to study several properties of the first- and second-order RNN's described above.

More recently we have been studying a more complex language represented by a much larger DFA in order to show how learning and generalization performance scale with the complexity of the problem. The DFA of this problem is a minimal, randomly generated 10-state machine referred to as "Random-10" in the following discussion. We present a some preliminary comparative results for this language as well.

These eight grammars generate regular languages with strings of arbitrary length over the alphabet $\Sigma = \{0, 1\}$, and can be described as follows:

$$
\begin{aligned}
T_1 &= 1^*. \\
T_2 &= (10)^*. \\
T_3 &= \text{an even number of consecutive 1's is always followed by}
\end{aligned}
$$

an even number of consecutive 0's.

T_4 = any string not containing "000".

T_5 = $[(01|10)(01|10)]^*$

T_6 = Mod-3 $[(N_1 - N_0) \bmod 3 = 0]$.

T_7 = $0^*1^*0^*1^*$.

Random-10 = randomly generated DFA; see Figure 2.

In this notation, (string)* represents a string repeated 0 or more times, (string-A|string-B) represents "either string-A or string-B" (exclusive-OR), and N_0, N_1 represent the number of 0's and 1's, respectively, contained in a string. (Note that our definition of T_5 is *not* the dual parity (4 state DFA) that is usually used. We gave our own interpretation of this language. For results on this language, see [17, 53].) In Figure 2 we show diagrams of the discrete finite state automata (DFA) that recognize these eight languages. These are the state automata that the neural networks of our study try to infer from example strings of their languages.

Except for T_6, all the Tomita languages become exponentially sparse with increasing length. That is, the fraction of positive examples of length L is exponentially smaller than the total number of examples of length L. This has the implication that with increasing length, the different states in the DFA have varying importance (and representation) in a sample of strings of that length. These effects can be balanced by choosing alphabetical training sets which have a uniform distribution over length. It has been shown empirically, via discrete methods [30], that a selection of examples which ranges over lengths $0,\ldots,D+5$ should be sufficient to infer a generating DFA of depth D. (The *depth* of an DFA is the depth of a graph constructed from it with the initial state at the root.) This hypothesis is also borne out in the case of recurrent neural nets, assuming the network has a sufficient number of weights.

Languages T_1, T_2, and T_6 have depth 1; T_3, T_4, and T_5 have depth 3; T_7 and Random-10 have depth 4. Hence we use strings up to length $D + 5 = 9$ in our training sets, as discussed below.

3.2 Extracting DFA's from Neural Networks

We briefly discuss a method for extracting a DFA from a recurrent neural network after (or during) training. Essentially the algorithm works by heuristically clustering the network's distribution in N-dimensional state space and then discretizing the space using staircase functions $S_i' = S_i - S_i \bmod (1/q)$, where $q \geq 2$ is an integer *quantization parameter*. For other methods of creating state transition diagrams, see [6, 54]. [Also, [54] gives another method for DFA extraction.] The resulting state automaton is an approximate discretized form of the neural network's dynamics and classification behavior. If the extracted DFA is equivalent to the

original DFA (*i.e.*, the one which generated the neural net's training set), the neural network has successfully inferred the grammar. We have shown previously [40] that these extracted DFA's often outperform the neural network in generalization power, even when the extracted DFA is not exactly correct. For more details of the extraction process, see [15].

4 Results

4.1 Description of Experiments

We present results that compare the training and generalization performance of first- and second-order recurrent neural networks on the eight benchmark grammars described above. Training simulations were performed for first- and second-order RNN's with 3 to 9 neurons. For each of these 14 configurations we ran 10 training simulations with data sets generated from each of these eight grammars. Each initial weight value was chosen randomly and uniformly over the interval $[-1.0, 1.0]$. Each network was then trained on all strings of length 0–9 in the target language (1,023 total), presented in alphabetical order. [Recently work by [47] showed enhanced learning capacity using alphabetical order in string presentation.] Training parameters were $\alpha = 0.5$, $\eta = 0.5$, $E_l = 0.2$, $E_h = 0.5$, $X_i = 50$, $X_r = 50$ (see Section 2.6 for definitions). Training proceeded for a maximum of 10 cycles of 500 epochs each.

Each network either converged on a solution in ≤ 5000 epochs or was terminated. The successfully converged networks were tested for generalization on all strings of length 10–15 (64,512 total). Additionally, we tested these networks on their DFA extraction performance. Results are presented below.

4.2 Convergence Results

We show the number of successful convergences (out of 10 runs) for each network configuration, and for those successful runs, the average convergence time measured in epochs, in Table 1. The latter are also plotted graphically in Figure 3. For comparison, we ran two set of runs on the Tomita languages: one which used bias terms and one which did not (all bias terms set to zero and frozen).

The general trend for both first and second order RNN's is faster convergence for larger numbers of neurons, and comparable relative convergence times for the two different orders. More complex grammars tend to require more time for convergence, though second order converges faster in most cases, especially with larger numbers of neurons. Also, the tabular data shows that first-order nets are less reliable, in terms of likelihood of convergence, particularly with small numbers of neurons, while second-order nets show no

degradation of training success except at the very smallest network size, $N = 3$. This behavior is much more pronounced in the case of the Random-10 language, where first-order RNN's consistently fail to converge in the required time, while second-order RNN's start to show consistent success for larger values of N.

Successful convergence occurred less often for networks that used bias (shown in the lower half of Table 1). Convergence times in successful cases were somewhat longer than for networks that did not use bias. Because biased networks converge less often than non-bias networks, we will not consider bias networks in further analysis.

Figure 3b shows the same convergence-time data, but plotted as a function of number of weights rather than neurons. This demonstrates a more pronounced effect: the convergence time drops quickly from a high value as the number of weights are increased, but appears to approach an asymptotic limit of efficiency. It should be remembered when viewing this plot that the *CPU time* required for these simulations is proportional to N^4L^2 in the second-order case (recall that N is the number of state neurons and L the number of input neurons) and to N^4 in the first order case (assuming large N), so although the "convergence time" (measured in epochs) drops to a constant as the networks grow, the CPU time required to compute the solutions grows (asymptotically) as N^4L^2 or N^4. If one were to plot the "real time" required for successful convergence, equal to the epochal time multiplied by cN^4L^2 or cN^4 (c = the average CPU time required for one epoch), there would be a minimum in the curve located near $N \approx 5$ (for these systems at least). This is an experimentally determined "optimal" network size.

4.3 Generalization Results

Generalization results are shown in Table 2 for all converged runs. Generalization was tested for strings of length 10–15 (64,512 strings) at tolerances $\epsilon = 0.2$ (all languages) and $\epsilon = 0.5$ (Tomita's languages only); the data shown gives the average *number* of strings in the test set that gave errors $E > \epsilon$. The minimal measure of performance of the RNN is $\epsilon = 0.5$; as long as the network holds its classification error below this level, we consider the RNN to perform correctly. However, a typical RNN will accumulate small inaccuracies in output classification, which we call *drift*, as it processes a string, and for sufficiently long strings, this accumulating inaccuracy can eventually lead to errors $E > 0.5$, which of course is a misclassification. One way to measure the extent to which a network is drifting is to look at the number of classification errors for $E > 0.2$. A network which has very few errors at this level will most likely perform extremely well for very long strings because its drift is very slight, whereas a network with more errors at $E > 0.2$ will have quicker

performance degradation at longer lengths.

For $\epsilon = 0.5$, both first- and second-order nets perform well. In all cases, the average number of errors is less than 1% of the test set. Both types of nets tested perfectly on languages T_1 and T_2; the worst performance was on T_5 and T_7. There is no consistent difference between first- and second-order RNN's at this level.

However, generalization at $\epsilon = 0.2$ gives a different story. Except for T_1 and T_2, which show near perfect test results, both architectures show considerable drift, as indicated by the large error rates (as much as 25%) shown in the tables. Two trends are immediately apparent in this data: (1) Second-order nets have considerably more drift than first-order nets. (2) Drift generally increases with larger numbers of neurons.

These trends both seem to indicate that a larger number of weights contributes to increased drift. This effect might be counteracted by (1) training to a smaller tolerance $\epsilon < 0.2$; (2) altering the discriminant function so it has steeper slope $[f(x) \rightarrow f(\beta x), \beta > 1.0]$; (3) selectively reducing the number of weights in the network (i.e. partial connectivity).

Figure 4 shows how convergence time and generalization performance are related. One point is plotted on this diagram for every successful training run, showing the number of epochs required to converge on one axis and the number of errors incurred on the generalization test for $\epsilon = 0.2$ on the other axis. From this figure we can conclude that quick convergence does not imply good generalization; in fact, convergence and generalization appear not to be correlated. Additionally, most runs converge within 1000 epochs; runs that converge in time > 1000 epochs do not have any notable performance advantage over more quickly convergent runs. Thus, for these training conditions and for languages of similar complexity, it makes sense to limit training time to 1000 epochs. If training appears to require longer time, then other parameters such as learning rate and momentum should probably be adjusted.

4.4 Inference (DFA Extraction) Results

Finally, we tested all successfully converged networks for successful inference by using the DFA extraction algorithm described above (Section 3.2). In Table 3 we show for each network configuration the average number of times a correct DFA was extracted from the corresponding neural network. The data are generated by running the extraction algorithm for quantization values $q = 2, \ldots, 10$ on each converged neural network, comparing the extracted DFA after Moore-minimization to the target DFA, and counting the number of matches. We compute the average number of matches for all 10 runs of each network configuration. In

addition, we show the average size of the smallest *unminimized* extracted DFA which minimizes to the correct target DFA.

The general trend is that the second-order architecture can extract the correct DFA with greater reliability. One notable exception to this trend is in language T_5, where first-order has the advantage, though both architectures perform relatively poorly in extracting the DFA for this language (most likely because T_5 is the largest DFA of the entire set of Tomita's grammars). On the whole, even though second-order architecture shows more drift inaccuracy, it is not enough to prevent it from inferring a correct grammar. (Though not shown, for most of the runs that converged the minimal extracted DFA for the random-10 runs was often the correct 10-state automaton - again all second-order network.)

The data shown for the average unminimized extracted DFA size show a strong trend: the size of the state machine is always larger when extracted from a first-order net. This appears to corroborate findings of Goudreau *et al.* [19] that indicate that first-order networks must use state-splitting to learn small regular languages. If indeed the first-order networks in the experiments reviewed here are using state-splitting, the extraction procedure will reveal this by producing a larger unminimized state machine. In fact, since state-splitting requires a larger fractional volume of state space to implement (assuming the same number of neurons and the same language to infer), this may well account for the higher failure rate for first-order nets in extracting correct DFA's.

5 Conclusions

We have shown that both first- and second-order recurrent neural networks can easily learn the grammars of Tomita using real-time recurrent learning using a full-gradient training algorithm but only second-order networks can learn a larger randomly generated 10-state DFA. We also show that networks of both architectures generalize well in recognizing longer input sequences than initially trained on, and can also extract the correct discrete finite automata (DFA) that generate the languages. Our results demonstrate that these properties hold fairly consistently for numbers of neurons in the range $N = 3, \ldots, 10$.

Both architectures have comparable learning power and generalization performance for simple regular grammars, *e.g.*, those in Tomita's benchmark. Second-order recurrent nets tend to be quicker and more reliable in converging to a good solution, particularly as the target language becomes more complex, as demonstrated by the convergence results for the random-10 language. However, second-order RNN's tend to accumulate more inaccuracies in classification when processing long strings. Thus, we speculate, second-

order recurrent nets will show poorer performance for longer input sequences. Our results demonstrate that this is indeed the case for input sequences of up to length 15. This problem with inaccuracy seems strongly correlated with larger numbers of weights. We propose a few methods to counteract this problem: (1) training to smaller tolerance, (2) using steeper threshold functions, and (3) selectively reducing the number of weights in the network. It is worth noting that if a training method is used that learns well with hard-thresholded neurons [56], many of the generalization inaccuracies will disappear all together.

Interestingly though, second-order nets appear more reliable for *inferring* the actual grammars, since they succeed more often in *extracting* correct DFA's. We hypothesize that the greater inference success of second-order nets is explained by the fact that first-order nets may need to use state-splitting [19] to implement state machines, which limits the flexibility and efficiency of the internal representations that first-order nets can use.

It is generally held that for every problem that neural networks could be applied to, there is an "optimal" size network best suited to that problem. Our results support this belief. We show that generalization on long strings is somewhat better for smaller networks of both architectures, but also that this trend is balanced by another factor – recurrent networks learn in fewer epochs the larger they are, and will altogether fail to converge on a solution if they are too small. It is still on open question as to how well first and second order nets compare on "very large" grammars. The trends imply that the second order nets would perform better, particularly in view of the extraction data presented above.

Finally, our initial experimental results with the more complex "Random-10" language indicate that second-order RNN's have *significantly better scaling behavior* than first-order RNN's in terms of their ability to infer more complicated grammars (those with larger number of states). For our convergence conditions *none* of the seventy first-order training runs converged whereas over half of the second-order did. Generalization results for these "unconverged" first-order runs indicated that these nets had learned essentially nothing and were giving random responses. Thus, on the basis of the results presented here, second-order nets appear to have significant computational advantages over simpler first-order architectures.

Why do second-order networks outperform the first-order variety? One interpretation is that second-order nets are a better neural network design for representing finite-state grammars. And it is well known that choosing the "right" representation for a problem makes a big difference in how easily that problem can be solved.

6 Acknowledgements

We would like to acknowledge the computational assistance of L.C. An, and useful discussions with D. Chen, M.W. Goudreau, K. Lang and C. Omlin. We also thank the reviewers of this paper for their helpful suggestions.

References

[1] D. Angluin, "On the complexity of minimum inference of regular sets," *Information and Control*, vol. 39, pp. 337–350, 1978.

[2] D. Angluin and C. Smith, "Inductive inference: Theory and methods," *ACM Computing Surveys*, vol. 15, no. 3, pp. 237–269, 1983.

[3] P. Ashar, S. Devadas, and A. Newton, *Sequential Logic Synthesis*. Norwell, MA: Kluwer Academic Publishers, 1992.

[4] B. Baird, "Associative memory in a simple model of oscillating cortex," in *Advances in Neural Information Processing Systems 2* (D. Touretzky, ed.), (San Mateo, CA), pp. 68–75, Morgan Kaufmann Publishers, 1990.

[5] P. Baldi and S. Venkatesh, "Number of stable points for spin-glasses and neural networks of higher orders," *Physical Review Letters*, vol. 58, no. 9, pp. 913–915, 1987.

[6] A. Cleeremans, D. Servan-Schreiber, and J. McClelland, "Finite state automata and simple recurrent recurrent networks," *Neural Computation*, vol. 1, no. 3, pp. 372–381, 1989.

[7] T. Cover, "Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition," *IEEE Trans on Electronic Computers*, vol. EC-14, p. 815, 1965.

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

[9] J. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.

[10] F. Fogelman-Soulie, Y. Robert, and M. Tchuente, eds., *Automata Networks in Computer Science, Theory and Applications*. Princeton, N.J.: Princeton University Press, 1987.

[11] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "A unified approach for integrating explicit knowledge and learning by example in recurrent networks," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 1, p. 811, IEEE 91CH3049-4, 1991.

[12] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "Unified integration of explicit rules and learning by example in recurrent networks," *IEEE Transactions on Knowledge and Data Engineering*, 1992. To appear.

[13] K. Fu, *Syntactic Pattern Recognition and Applications*. Englewood Cliffs, N.J: Prentice-Hall, 1982.

[14] C. Giles and T. Maxwell, "Learning, invariance, and generalization in high-order neural networks," *Applied Optics*, vol. 26, no. 23, p. 4972, 1987.

[15] C. Giles, C. Miller, D. Chen, H. Chen, G. Sun, and Y. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Computation*, vol. 4, no. 3, p. 380, 1992.

[16] C. Giles and C. Omlin, "Inserting rules into recurrent neural networks," in *Neural Networks for Signal Processing II, Proceedings of The 1992 IEEE Workshop* (S. Kung, F. Fallside, J. A. Sorenson, and C. Kamm, eds.), pp. 13–22, IEEE Press, 1992.

[17] C. Giles, G. Sun, H. Chen, Y. Lee, and D. Chen, "Higher order recurrent networks & grammatical inference," in *Advances in Neural Information Processing Systems 2* (D. Touretzky, ed.), (San Mateo, CA), pp. 380–387, Morgan Kaufmann Publishers, 1990.

[18] E. Goles and S. Martinez, *Neural and Automata Networks*. Boston, Mass.: Kluwer Academic Publishers, 1990.

[19] M. Goudreau, C. Giles, S. Chakradhar, and D. Chen, "First-order vs. second-order single layer recurrent neural networks," *IEEE Transactions on Neural Networks*, 1993. Accepted for publication.

[20] I. Guyon, L. Personnaz, J. Nadal, and G. Dreyfus, "Storage and retrieval of complex sequences in neural networks," *Physical Review A*, vol. 38, no. 12, p. 6365, 1992.

[21] L. Guyon, V. Vapnik, B. Boser, L. Bottou, and S. Solla, "Structural risk minimization for character recognition," in *Advances in Neural Information Processing Systems 4* (J. Moody, S. Hanson, and R. Lippmann, eds.), (San Mateo, CA), pp. 471–479, Morgan Kaufmann Publishers, 1992.

[22] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation*. Redwood City, CA: Addison-Wesley Publishing Company, Inc., 1991.

[23] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1979.

[24] J. Hopfield and D. Tank, "Neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, pp. 141–152, 1985.

[25] B. Horne, D. Hush, and C. Abdallah, "The state space recurrent neural network with application to regular grammatical inference," Tech. Rep. UNM Technical Report No. EECE 92-002, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM, 87131, 1992.

[26] M. Jordan, "Attractor dynamics and parallelism in a connectionist sequential machine," in *Proceedings of the Ninth Annual conference of the Cognitive Science Society*, pp. 531–546, Lawrence Erlbaum, 1986.

[27] S. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. Shannon and J. McCarthy, eds.), pp. 3–42, Princeton, N.J.: Princeton University Press, 1956.

[28] C. Koch and T. Poggio, "Multiplying with synapses and neurons," in *Single Neuron Computation* (S. Z. T. McKenna, J. Davis, ed.), ch. 12, Boston, MA: Academic Press, Inc., 1992.

[29] Z. Kohavi, *Switching and Finite Automata Theory*. New York, NY: McGraw-Hill, Inc., second ed., 1978.

[30] K. Lang, "Random dfa's can be approximately learned from sparse uniform examples," in *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, (New York, N.Y.), pp. 45–52, ACM, 1992.

[31] Y. Lee, G. Doolen, H. Chen, G. Sun, T. Maxwell, H. Lee, and C. Giles, "Machine learning using a higher order correlational network," *Physica D*, vol. 22-D, no. 1-3, pp. 276–306, 1986.

[32] S. Lucas and R. Damper, "Syntactic neural networks," *Connection Science*, vol. 2, pp. 199–225, 1990.

[33] W. McCulloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[34] L. Miclet, "Grammatical inference," in *Syntactic and Structural Pattern Recognition; Theory and Applications* (H. Bunke and A. Sanfeliu, eds.), ch. 9, Singapore: World Scientific, 1990.

[35] M. Minsky, *Computation: Finite and Infinite Machines*, ch. 3, pp. 32–66. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1967.

[36] M. Minsky and S. Papert, *Perceptrons*. Cambridge, MA: MIT Press, 1969.

[37] M. Mozer and J. Bachrach, "Discovering the structure of a reactive environment by exploration," *Neural Computation*, vol. 2, no. 4, pp. 447–457, 1990.

[38] K. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Trans. on Neural Networks*, vol. 1, no. 1, p. 4, 1990.

[39] C. Omlin and C. Giles, "Training second-order recurrent neural networks using hints," in *Proceedings of the Ninth International Conference on Machine Learning* (D. Sleeman and P. Edwards, eds.), (San Mateo, CA), pp. 363–368, Morgan Kaufmann Publishers, 1992.

[40] C. Omlin, C. Giles, and C. Miller, "Heuristics for the extraction of rules from discrete-time recurrent neural networks," in *Proceedings International Joint Conference on Neural Networks 1992*, vol. I, pp. 33–38, June 1992.

[41] Y. Pao, *Adaptive Pattern Recognition and Neural Networks*. Reading, MA: Addison-Wesley Publishing Co., Inc., 1989.

[42] S. Perantonis and P. Lisboa, "Translation, rotation, and scale invariant pattern recognition by higher-order neural networks and moment classifiers," *IEEE Transactions on Neural Networks*, vol. 3, no. 2, p. 241, 1992.

[43] P. Peretto and J. Niez, "Long term memory storage capacity of multiconnected neural networks," *Biological Cybernetics*, vol. 54, p. 53, 1986.

[44] L. Personnaz, I. Guyon, and G. Dreyfus, "High-order neural networks: Information storage without errors," *Europhysics Letters*, vol. 4, p. 863, 1987.

[45] J. Pollack, "The induction of dynamical recognizers," *Machine Learning*, vol. 7, pp. 227–252, 1991.

[46] J. Pollack, "Recursive distributed representations," *Journal of Artificial Intelligence*, vol. 46, p. 77, 1990.

[47] S. Porat and J. Feldman, "Learning automata from ordered examples," *Machine Learning*, vol. 7, no. 2-3, p. 109, 1991.

[48] D. Psaltis, C. Park, and J. Hong, "Higher order associative memories and their optical implementations," *Neural Networks*, vol. 1, p. 149, 1988.

[49] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*, ch. 8, Cambridge, MA: MIT Press, 1986.

[50] H. Siegelmann and E. Sontag, "Turing compatability with neural nets," *Applied Mathematics Letters*, vol. 4, no. 6, pp. 77–80, 1991.

[51] G. Sun, H. Chen, C. Giles, Y. Lee, and D. Chen, "Connectionist pushdown automata that learn context-free grammars," in *Proceedings of the International Joint Conference on Neural Networks 1990* (M. Caudill, ed.), vol. I, (Hillsdale, N.J.), pp. 577–580, Lawerence Erlbaum, 1990.

[52] M. Tomita, "Dynamic construction of finite-state automata from examples using hill-climbing," in *Proceedings of the Fourth Annual Cognitive Science Conference*, (Ann Arbor, Mi), pp. 105–108, 1982.

[53] R. Watrous and G. Kuhn, "Induction of finite-state languages using second-order recurrent networks," *Neural Computation*, vol. 4, no. 3, p. 406, 1992.

[54] R. Watrous and G. Kuhn, "Induction of finite state languages using second-order recurrent networks," in *Advances in Neural Information Processing Systems 4* (J. Moody, S. Hanson, and R. Lippmann, eds.), (San Mateo, CA), pp. 309–316, Morgan Kaufmann Publishers, 1992.

[55] R. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

[56] Z. Zeng, R. Goodman, and P. Smyth, "Learning finite state machines with self-clustering recurrent networks," *Neural Computation*, 1993. to be published.

Table 1. Convergence results for the seven Tomita languages and the random-10 language. Data are for first- and second-order RNN's with 3 to 9 neurons: (top) without bias; (bottom) with bias [random-10 was not run with bias]. Ten runs were generated for each architecture, size, and language. The left subcolumn for each value N shows the number of runs (out of 10) which converged within 5000 epochs. The right subcolumn shows the average number of epochs for those runs which did converge.

lang	arch	$n=3$		$n=4$		$n=5$		$n=6$		$n=7$		$n=8$		$n=9$	
T-1	F.	10	40.4	10	40.0	10	32.5	10	27.7	10	28.8	10	26.4	10	23.1
	S.	10	46.3	10	37.2	10	37.7	10	42.1	10	33.3	10	34.3	10	29.9
T-2	F.	10	203.7	10	175.2	10	162.8	10	147.2	10	134.2	10	126.8	10	113.2
	S.	10	183.3	10	113.6	10	108.0	10	101.0	10	80.7	10	85.4	10	77.7
T-3	F.	3	244.3	9	197.2	10	406.4	10	328.0	10	193.5	10	459.2	10	425.6
	S.	10	294.7	10	94.7	10	64.3	10	62.8	10	54.5	10	53.8	10	51.7
T-4	F.	9	211.1	9	103.4	10	95.5	10	93.7	10	97.0	10	83.3	10	100.6
	S.	10	133.7	10	81.5	10	70.8	10	55.1	10	54.4	10	49.0	10	46.2
T-5	F.	1	591.0	4	490.7	10	470.5	10	376.4	10	327.9	10	277.1	10	258.7
	S.	2	1728.0	10	491.9	10	289.8	10	265.2	10	213.4	10	190.5	10	158.6
T-6	F.	4	1311.5	7	445.6	7	364.4	8	344.1	7	369.6	6	346.0	10	506.2
	S.	10	95.3	10	70.3	10	64.4	10	57.8	10	54.1	10	50.1	10	49.6
T-7	F.	3	792.3	4	427.7	6	438.8	9	454.7	7	283.6	8	306.2	10	342.6
	S.	8	570.6	10	306.5	10	214.9	10	163.9	10	160.2	10	147.5	10	121.0
R-10	F.	0	–	0	–	0	–	0	–	0	–	0	–	0	–
	S.	0	–	0	–	1	377.0	3	458.3	8	556.6	9	731.8	10	202.9
T-1	F.B.	10	37.0	10	34.1	10	29.0	10	27.5	10	26.2	10	25.8	10	25.3
	S.B.	10	38.9	10	42.7	10	33.7	10	35.1	10	31.9	10	34.1	10	32.2
T-2	F.B.	10	188.1	10	167.5	10	152.6	10	143.6	10	120.9	10	119.5	10	93.6
	S.B.	10	146.7	10	150.9	10	113.7	10	113.0	10	97.0	10	81.5	10	88.2
T-3	F.B.	5	119.8	9	344.7	9	957.3	8	458.6	10	498.2	10	493.1	9	365.1
	S.B.	8	119.1	10	145.2	10	79.4	10	67.2	10	62.0	10	57.1	10	51.8
T-4	F.B.	7	180.7	10	153.0	9	105.9	9	135.8	9	91.1	10	85.2	10	91.2
	S.B.	10	211.2	10	94.2	10	100.3	10	80.5	10	71.9	10	60.4	10	58.6
T-5	F.B.	1	886.0	2	460.5	7	391.0	8	482.5	9	295.6	9	251.4	10	288.7
	S.B.	6	588.2	9	364.9	10	290.8	10	271.6	10	191.1	10	211.0	10	177.7
T-6	F.B.	0	–	5	235.0	9	275.4	9	210.8	9	362.2	7	161.1	9	211.4
	S.B.	10	306.4	10	75.7	10	77.7	10	61.2	10	56.0	10	53.2	10	57.2
T-7	F.B.	0	–	5	389.8	7	890.6	3	343.3	5	338.4	7	237.0	9	493.4
	S.B.	5	880.8	10	574.7	10	265.4	9	193.2	10	176.5	10	150.2	10	197.8

Table 2. Generalization test results for first- and second-order RNN's. Tests were run only on nets that converged. The tables show the average number of errors made over the test set of all strings of length 10–15 (64,512 total) in the learned language. (Top) results for tolerance $\epsilon = 0.2$; (bottom) for $\epsilon = 0.5$. [Preliminary results only ($\epsilon = 0.2$) for the Random-10 language.]

lang	arch	n=3	n=4	n=5	n=6	n=7	n=8	n=9
T_1	F.	1	3	1	2	2	3	1
T_1	S.	3	2	3	3	2	3	1
T_2	F.	6	2	2	3	4	2	2
T_2	S.	2	6	2	9	5	3	5
T_3	F.	66	85	85	181	285	251	219
T_3	S.	682	908	933	2101	1202	2191	1736
T_4	F.	38	39	117	87	87	88	496
T_4	S.	340	147	134	592	369	791	1240
T_5	F.	1048	191	207	443	263	257	345
T_5	S.	984	655	1060	815	917	635	696
T_6	F.	39	239	510	511	830	720	896
T_6	S.	8130	7602	16254	13045	8248	12192	8725
T_7	F.	170	196	285	366	711	126	717
T_7	S.	419	694	1121	1307	1292	388	889
R_{10}	F.	–	–	–	–	–	–	–
R_{10}	S.	–	–	22	18	23	26	18
T_1	F.	0	0	0	0	0	0	0
T_1	S.	0	0	0	0	0	0	0
T_2	F.	0	0	0	0	0	0	0
T_2	S.	0	0	0	0	0	0	0
T_3	F.	23	0	1	9	97	36	40
T_3	S.	27	0	0	2	0	3	22
T_4	F.	13	8	29	2	18	26	90
T_4	S.	5	0	0	19	32	94	41
T_5	F.	105	58	30	149	56	34	12
T_5	S.	429	146	261	145	150	99	125
T_6	F.	5	12	0	9	58	3	288
T_6	S.	0	0	74	10	0	30	0
T_7	F.	27	12	63	75	169	37	317
T_7	S.	95	132	245	110	274	22	58

Table 3. DFA Extraction Analysis. This table presents data for the heuristic DFA extraction procedure (described in the text) as applied to all successfully converged first- and second-order runs in this study. Each column gives data for different numbers N of neurons. The left subcolumn in each column gives the average number of successful extractions. The right subcolumn gives the average smallest uminimized size of correct extracted DFA's.

		$n=3$		$n=4$		$n=5$		$n=6$		$n=7$		$n=8$		$n=9$	
lang	arch	extr	size	extr	size	extr	size	extr	size	extr	size	extr	size	extr	size
T_1	F.	9.0	3.9	9.0	4.8	9.0	5.6	9.0	6.3	9.0	7.7	9.0	7.1	9.0	9.2
T_1	S.	9.0	3.1	9.0	3.6	9.0	5.0	9.0	5.4	9.0	7.1	9.0	7.2	9.0	8.6
T_2	F.	9.0	5.9	8.4	9.7	8.7	9.3	8.9	9.5	8.7	12.0	8.7	14.7	8.9	13.3
T_2	S.	9.0	3.5	9.0	4.3	8.9	6.0	9.0	6.6	9.0	7.8	9.0	10.1	9.0	9.9
T_3	F.	2.4	8.5	7.3	10.3	7.8	12.2	7.7	11.2	7.0	15.6	6.0	16.2	6.7	19.4
T_3	S.	7.0	9.2	8.5	7.4	8.8	8.4	6.9	9.0	7.5	10.2	4.9	11.8	4.9	12.5
T_4	F.	6.8	6.9	7.6	7.0	8.7	8.0	8.5	8.4	8.3	12.7	7.2	12.1	6.1	15.2
T_4	S.	8.5	6.3	8.7	7.9	8.7	7.5	8.6	7.8	8.2	9.3	7.8	12.7	7.3	12.3
T_5	F.	0.7	10.0	2.0	16.5	5.6	16.4	4.2	25.9	4.4	25.2	3.0	28.7	3.8	41.6
T_5	S.	0.5	15.0	2.9	13.2	2.1	14.9	1.7	23.5	2.0	16.6	1.4	30.4	1.5	24.3
T_6	F.	3.6	5.8	5.6	9.3	6.1	8.8	6.4	11.3	5.0	15.4	4.5	13.4	5.7	15.0
T_6	S.	9.0	3.1	8.8	3.2	8.4	4.5	7.8	5.4	8.4	6.6	5.8	8.9	6.3	10.5
T_7	F.	2.7	6.0	3.4	8.8	4.9	9.0	7.6	11.2	5.4	13.4	6.7	13.0	6.4	15.1
T_7	S.	6.3	7.4	8.2	7.3	7.8	7.6	8.1	8.5	7.3	9.0	8.3	10.7	8.1	10.7

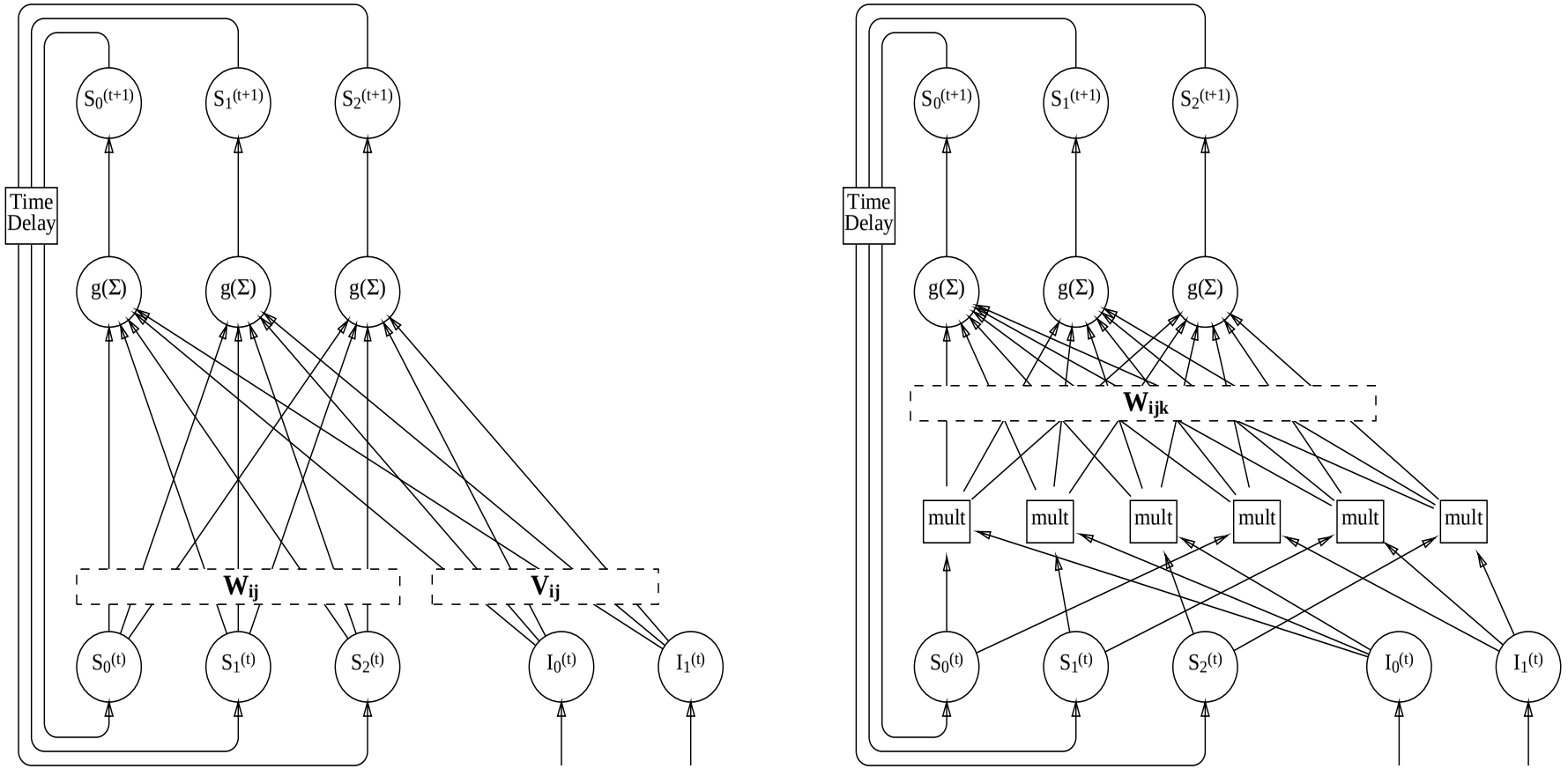

Figure 1. First-order (left) and second-order (right) single layer recurrent neural networks. $S_i^{(t)}$ represents the value of the i^{th} state neuron at time t, and $I_k^{(t)}$ represents the value of the k^{th} input neuron at time t. Units marked ‘‘g(Σ)’’ represent the sigmoid function operating on the sum of all incoming connections. Blocks marked ‘‘mult’’ represent the second-order operation $W_{ijk} \times S_j^{(t)} \times I_k^{(t)}$.

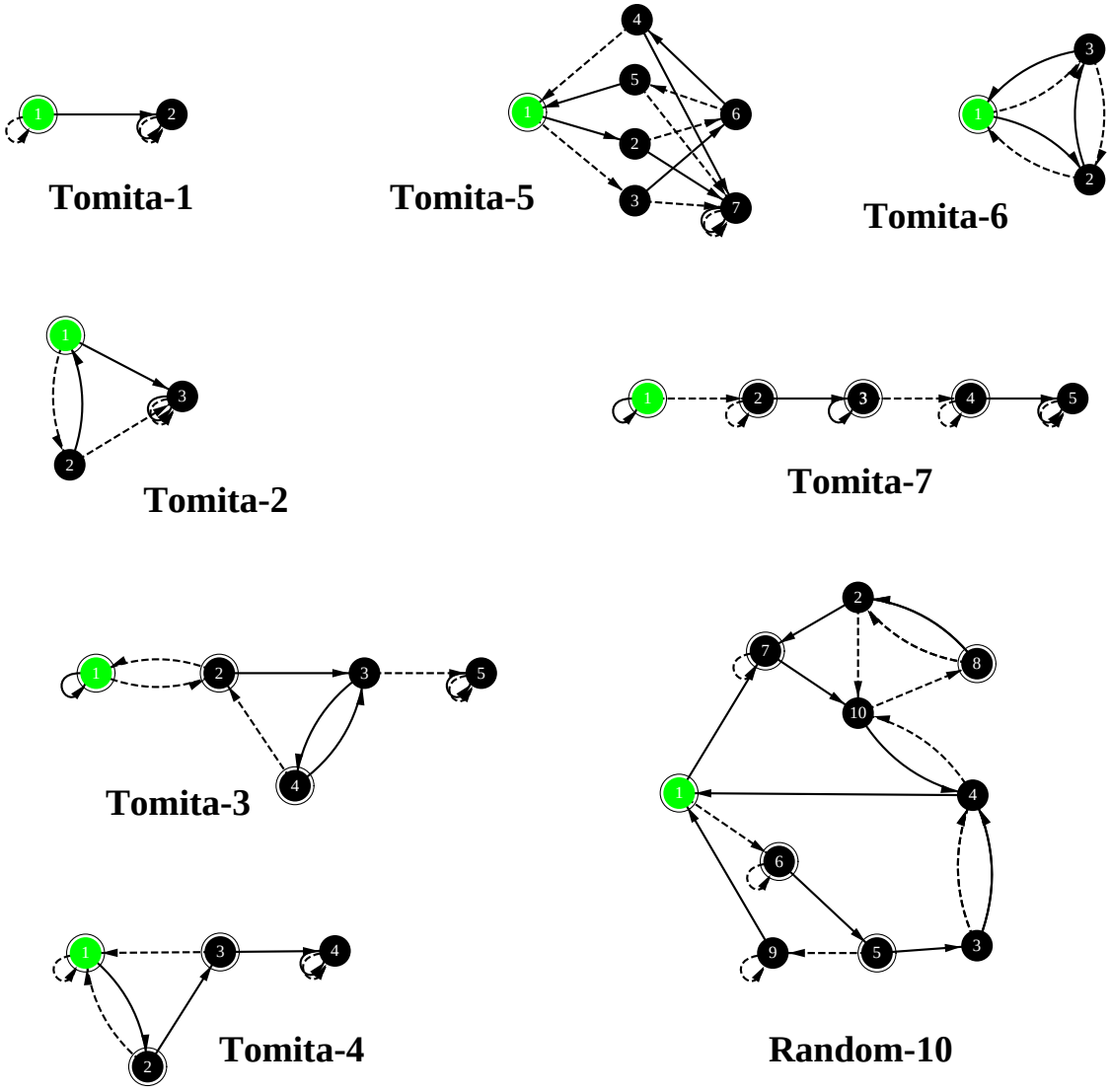

Figure 2. Shown here are the discrete finite state automata (DFA) for the seven Tomita languages and the random-10 language described in the text. The initial state in each DFA is shaded gray and labelled ‘‘1’’. Final (accepting) states are drawn with an extra circle. Solid arcs indicate transitions for input symbol 0; dashed arcs, symbol 1.

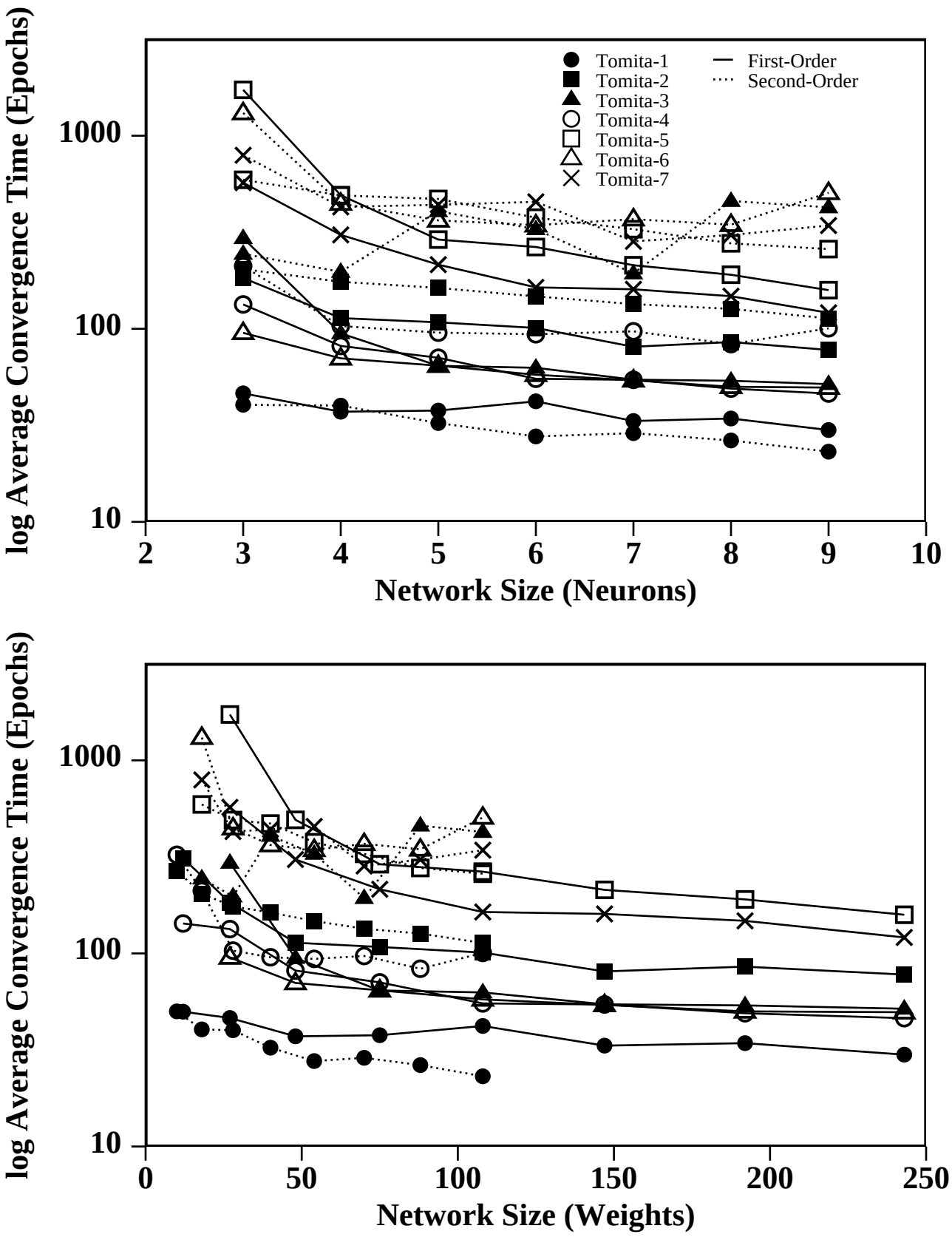

Figure 3. Average convergence times (in epochs) of first- and second-order RNN's for each of Tomita's seven languages: (top) dependence on number of neurons; (bottom) dependence on number of weights.

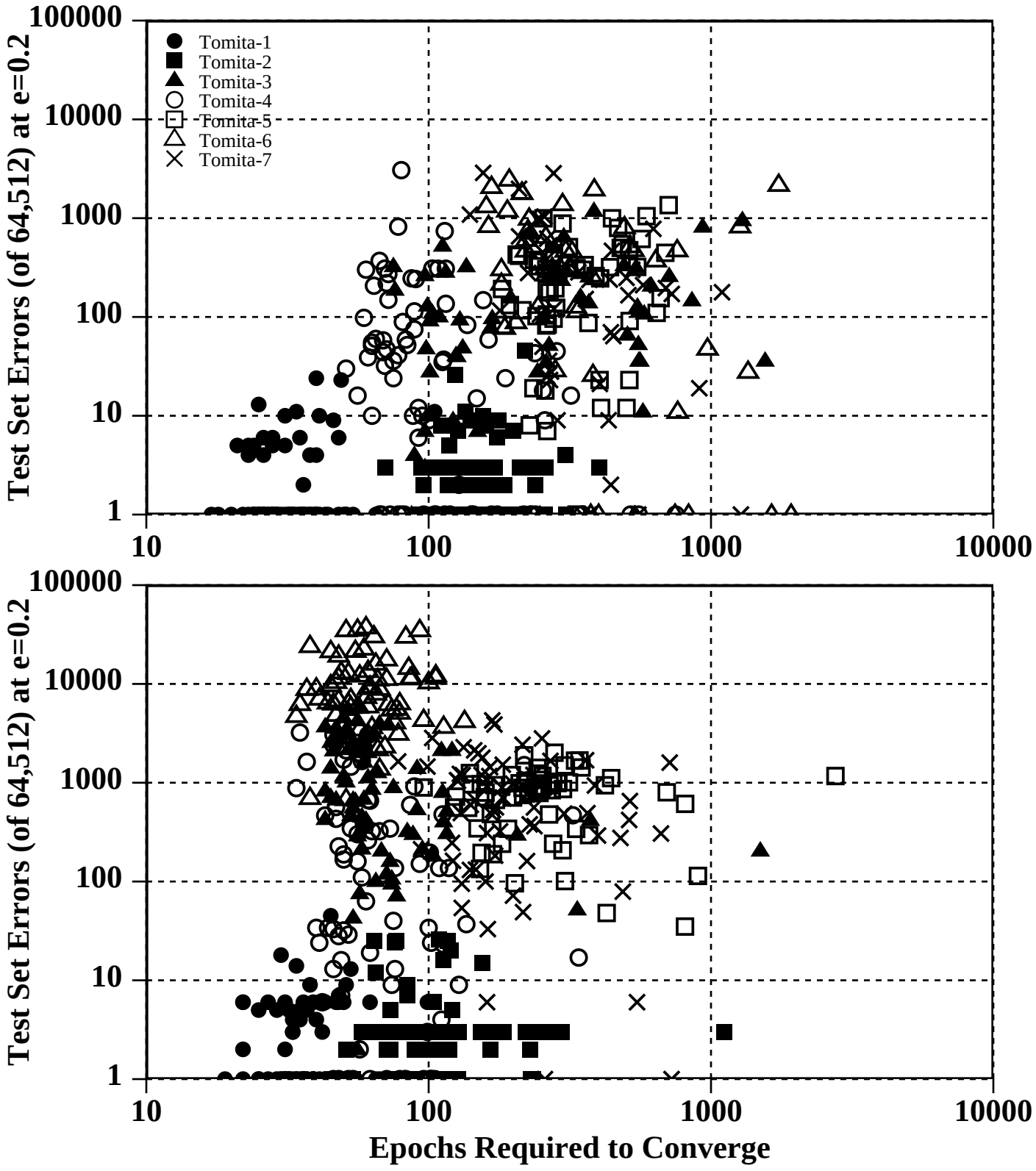

Figure 4. Convergence times (in epochs) for each successful training run, correlated with the number of errors made during generalization tests for E>0.2. (top) first-order, (bottom) second-order. See text (Section 4.3) for further explanation.