Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks

Janet Wiles
Departments of Computer Science and Psychology
University of Queensland
Queensland 4072, Australia
janetw@cs.uq.oz.au

Jeff Elman
Department of Cognitive Science, 0515
University of California, San Diego
La Jolla, California 92093-0515
elman@cogsci.ucsd.edu

Abstract

The broad context of this study is the investigation of the nature of computation in recurrent networks (RNs). The current study has two parts. The first is to show that a RN can solve a problem that we take to be of interest (a counting task), and the second is to use the solution as a platform for developing a more general understanding of RNs as computational mechanisms. We begin by presenting the empirical results of training RNs on the counting task. The task ($a^n b^n$) is the simplest possible grammar that requires a PDA or counter. A RN was trained to predict the deterministic elements in sequences of the form $a^n b^n *$ where n=1 to 12. After training, it generalized to n=18. Contrary to our expectations, on analyzing the hidden unit dynamics, we find no evidence of units acting like counters. Instead, we find an oscillator. We then explore the possible range of behaviors of oscillators using iterated maps and in the second part of the paper we describe the use of iterated maps for understanding RN mechanisms in terms of "activation landscapes". This analysis leads to used an understanding of the behavior of network generated in the simulation study.

Introduction

It is common to view the brain as a computer. But the real question is, What sort of computer might the brain be?

One reasonable assumption is that the functionally, brain computation can be understood within the framework of discrete finite automata (DFA). One can then use the Chomsky Hierarchy as a tool for inferentially classifying the computational power of the brain. If brains produce behaviors which fall entirely within the realm of context-free grammars, for example, we might suppose that the brain is the computational equivalent of a Linear Bounded Automata (since this class of machines is both necessary and sufficient for the generation and recognition of such languages).

In reality, however, formal analysis of behavior does not suggest that such a neat typing will be possible. Furthermore, in the past decade, it has been suggested that the brain may not be best modeled as a type of DFA, but rather as a continuous analog automaton of the sort represented by neural networks.

This possibility then raises the question, What sort of computers are neural networks? Attempts to answer this question generally attempt either to probe capacity through empirical experimentation (e.g., of the sort reported in Cleeremans, Servan-Schreiber, & McClelland, 1989; Elman, 1991; Giles, Miller, Chen, Chen, Sun, & Lee, 1992; Manolios & Fanelli, 1994; Watrous & Kuhn, 1992) or else to establish theoretical capacity through formal analysis (Kolen, 1994; Pollack, 1991; Seligmann & Sontag, 1992).

Lacking in much of this work, however, has been a close-grained analysis of the precise mechanisms which can be employed by neural networks in the service of specific computational tasks. Elman (1991), for example, demonstrates the ability of a recurrent network to emulate certain aspects of a pushdown automaton (namely, to process recursively embedded structures to a limited depth); the analysis of this network suggests that the network partitions the hidden unit state space to represent grammatical categories and depth of embedding. The network weights then implement a dynamics which allow the network to move through this state space in a rule-following manner which is consistent with the context-free grammar that produces the input strings. This analysis is suggestive at best, however, and leaves many important questions unanswered. How does the RN solution compare with that of a stack in terms of processing capacity? Are the solutions functionally exactly equivalent or are there differences? Are these differences relevant to understanding cognitive behaviors? If RNs are dynamical systems, then how can dynamics be employed to carry out specific computational tasks?

Our goal in the project which this work initiates is to redress this failing. We wish to investigate the nature of computation in recurrent networks by discovering the detailed mechanisms which are employed to carry out specific computational requirements. The strategy is first to train a recurrent network to produce a behavior which is of a priori interest, and which has known a computational solution within the realm of DFA. We then analyze the recurrent network to discover whether the solution is

equivalent to that of the DFA or whether it is different. If the solution is different, the question then becomes whether the solution is more or less likely to provide insight into the computational mechanisms employed by the brain in the service of cognitive behaviors.

Simulation

The work of Giles and colleagues has demonstrated that recurrent networks (RNs) can provide reasonable approximations of the simplest class of DFAs: Finite State Automata (FSA). The network solution appears to involve an instantiation of the state space required by the FSA, although there are interesting and possibly useful differences. (For example, path information tends to be saved—gratuitously—in the RN, and the RN state space probably has an intrinsic semantics whereas the topology of the FSA is, aside from the state transitions, undefined.) Our interest is therefore in exploring behaviors which require the next highest category of DFA, namely context-free languages. These require a form of pushdown automaton known as a Linear Bounded Automaton.

Task and stimuli

One of the simplest CF languages one can devise is the language $a^n b^n$; that is, the language consisting of strings of some number of *a*s followed by the same number of *b*s. This language requires a pushdown store in order to keep track of the *a*s in order that each *a* may be matched by a later *b*. In reality, though, the full power of the store is not really essential. All that is required is a counter.

We generated a training set consisting of 356 strings, containing a total of 2,298 tokens of *a* and *b*. These strings conformed to the form $a^n b^n$, with *n* ranging from 1 to 11 (meaning strength length varied from 2 to 22). Length was biased toward shorter strings (e.g., there were 129 strings of depth 1 and 7of depth 11).

A separate set of test stimuli were generated which consisted of all possible strings with *n* ranging from 1 to 30 so that we might test generalization to depths greater than that encountered during training.

The network's task was to take a symbol as its input and to predict the next input. Successful performance would require that the network predict an initial *a* (since all strings begin with this token); the network should then predict *a* or *b* with equiprobability until the first *b* is encountered. The network should then predict *b* for *n* timesteps, where *n* equals the number of *a*s that were input. Following that, the network should then predict an *a* to indicate the end of the old string and beginning of the next string.

Network and training

20 recurrent networks of the form show in Figure 1 was used. There were two input units (representing the two

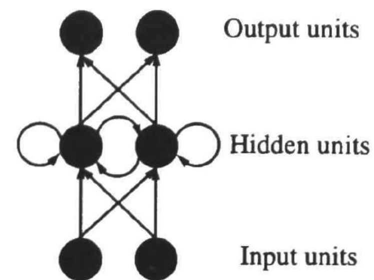


Figure 1: Network architecture

possible inputs, *a* and *b*) and two output units (representing the network's predictions for the next inputs). Two hidden units were connected with full recurrence. Networks were initialized with different random weights.

Networks was trained using back propagation through time (for 8 time steps). Training was carried out for a total of 3 million inputs.

Results

The networks' performance was evaluated using the test data in which all strings from depth 1 to 30 were present. Testing was carried out following 1, 2, and 3 million training cycles.

After 1 million training cycles 9 of the networks learned the language $a^n b^n$ for $n \sim 7$. One network generalized to n=11. The other networks learned the language $a^* b^*$. This is the language consisting of any number of *a*s followed by any number of *b*s; in this case the network simply predicts its input.

After 2 million training cycles, 4 of the 20 networks generalized the correct language to n=12. One network generalized to n=18. Remaining networks had learned $a^* b^*$.

After 3 million training cycles, no networks were successful for n>11. Those networks which had exhibited generalization at earlier stages of learning lost their solution and in many cases reverted to $a^* b^*$.

Subsequent replications on additional groups of 20 networks yield essentially the same statistics, including at least one network which generalizes to approximately a depth of 18. We therefore focus on this network for analysis.

Analysis: Part I

Our first conjecture was that the network might have solved this task by employing one or both of its hidden units as counters. This would be indicated by that hidden

unit's activation function changing (e.g., increasing) as a monotonic function of the number of *a* inputs. The magnitude of the final activation state of the unit would therefore encode *n*.

However, when we plotted the hidden unit activations as a function of input we saw nothing which resembled a counter. Instead, to our surprise, we found both units oscillating in activation value over time (but not in synchrony). We took this as prima facie evidence that the counter hypothesis was falsified and then attempted to construct another hypothesis. This involved stepping back and considering in more general terms what sorts of dynamical behaviors might be generated in recurrent networks under very limiting conditions.

Dynamics in recurrent networks

Let us consider a simpler version of the network used in the simulation; this network will have two inputs, two outputs, a bias, and a single hidden unit with a self-recurrent connection. This network is shown in Figure 2.

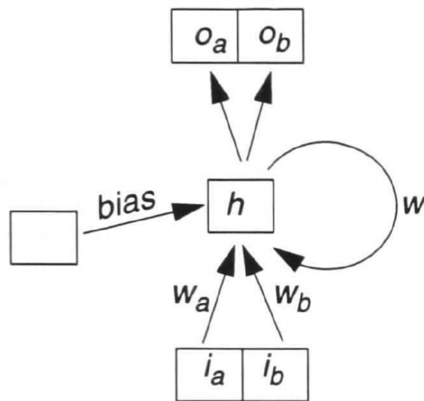


Figure 2: Network used in analysis

We are interested in the dynamics of the hidden unit, *h*, under various conditions. This unit has several sources of input: input tokens, bias, and self-recurrence. We begin by recognizing that when the input is held constant (as when the network is processing a string of *a*s which it has to count), then the only thing which changes is the self-recurrent excitation. We can therefore subsume all other inputs under a bias term:

$$netinput = \underbrace{bias + i_a w_a + i_b w_b}_{b} + h(t-1)\,w$$
$$netinput = b + h(t-1)\,w$$

The activation function for this unit is then

$$h(t) = \frac{1}{1 + e^{-(wh(t-1) + b)}}$$

If we let $w = 10$ and $b = -5$, then we observe the unit has the properties shown in Figure 3. The unit has 3 fixed points. Thus, over time, if we begin with h(0) greater than 0.5, we see the movement in activation space shown in Figure 4.

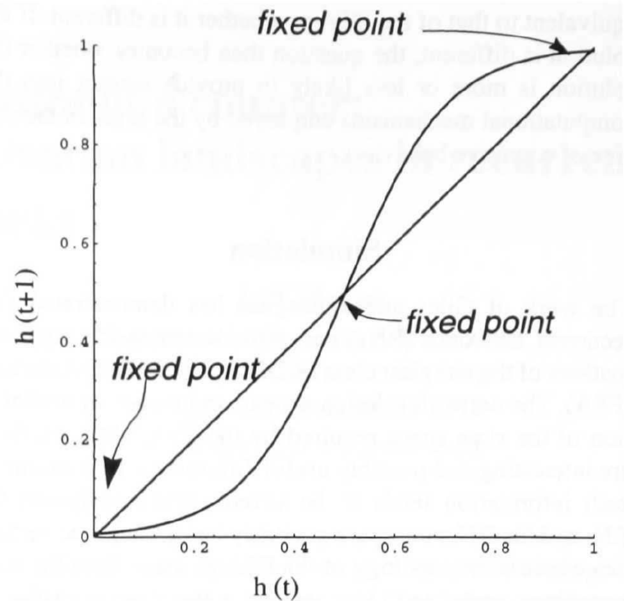


Figure 3: Dynamical properties of network shown in Figure 2, with w=10 and b=-5

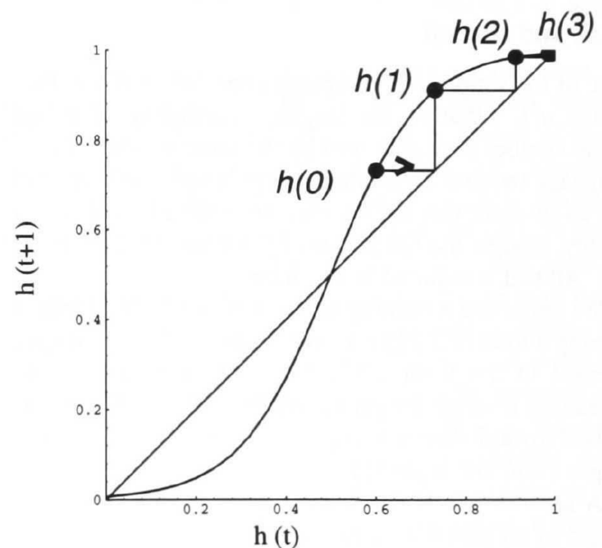
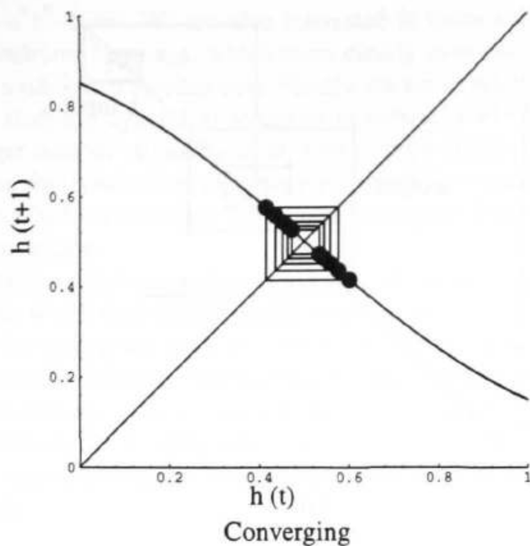


Figure 4: Convergence properties of network in Figure 2

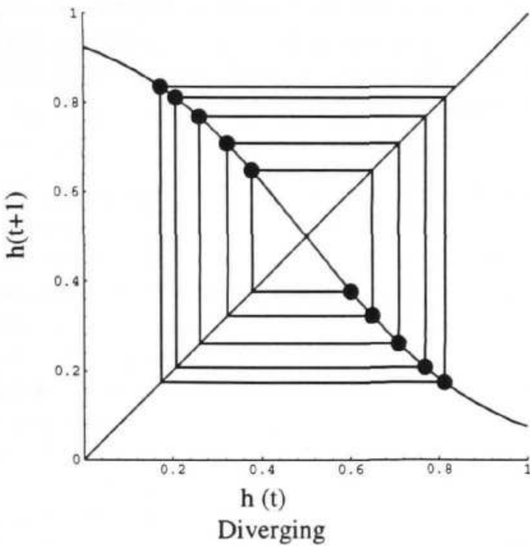
Suppose the sign of the self-recurrent weight is negative. In effect, this flips the activation function and changes the convergence properties. We now find that we have fixed points as before, but we oscillate back and forth above and below the middle fixed point. Depending on the steepness of the slope and our initial value, we either diverge out or converge inward. This is shown in Figure 5.

Finally, we note that if we could change the slope of the hidden unit's activation function dynamically (i.e., during processing), then we could produce two regimes, e.g., first converging and then diverging. This is shown in Figure 6.

We now ask how such behavior might be useful to us?



Converging



Diverging

Figure 5: Oscillating behavior found with negative self-recurrent weight.

In Figure 6 we see the activation function of the single hidden unit in the network in Figure 2, first when the slope is shifted to the left, and then when the activation function is shifted to the right. In both cases the hidden unit computes an iterated map on itself, given a constant input.

Let us imagine that we have two hidden units instead of one, so that these two slopes represent different units. Further, let us imagine that the graph shown on the left is the first hidden unit's response to a series of *a* inputs. The unit's activation will converge on successive iterations; how far it converges depends on how many iterations with the same input we carry out. Now let us assume that the input changes to *b*. The graph on the right might represent the iterated map on the second hidden unit. The initial starting point depends on the final state value of the first hidden unit; it then diverges outward.

To make use of this for a counting task, we need two more

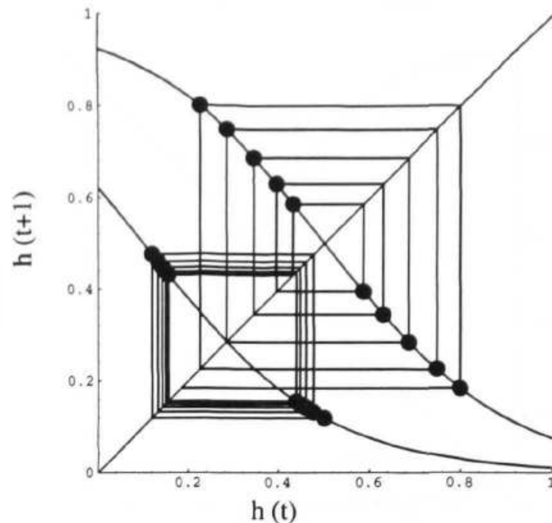


Figure 6: Converging oscillations followed by diverging oscillations

things to be true. First, we need output units which can implement a hyperplane on the divergence phase so that the network can establish a criterial value which will signal the end of sequence. Second, during the initial phase, while the first hidden unit is converging, we would like to have the second hidden unit (rightmost graph) "out of the way"; this could be accomplished if the input it receives from the first hidden unit shifts the slope to its asymptotic region. Then, during the second phase, while the second hidden unit is diverging, we would like to have the first hidden unit suppressed in a similar way. Let us return to the actual simulation to see if this is what happens.

Analysis: Part II

Returning to the trained network shown in Figure 1, we can graph the activation function of each hidden unit under conditions when a sequence of *a*s are received, and when a sequence of *b*s are received. Figure 7 shows the activation function of the first hidden u nit during presentation of *a*s (rightmost plot) and *b*s (leftmost plot). Figure 8 shows the activation functions of the second hidden unit under similar conditions.

What we see is that each unit is "on" (i.e., has an activation function which is capable of producing discriminably different outputs) only during one type of input, and each unit responds to a different input. While one unit is active, it shuts off the other unit. When the input sequence switches from *a* to *b*, the other unit becomes active and shuts the first unit off.

Now let us look at the actual pattern of responses while this network processes a real sequence. This is shown in Figure 9. Here we see exactly the desired behavior: One hidden unit in essence "winds up" like a spring as it counts successive *a* inputs; when the first *b* is encountered, the second unit "unwinds" for exactly the amount of time

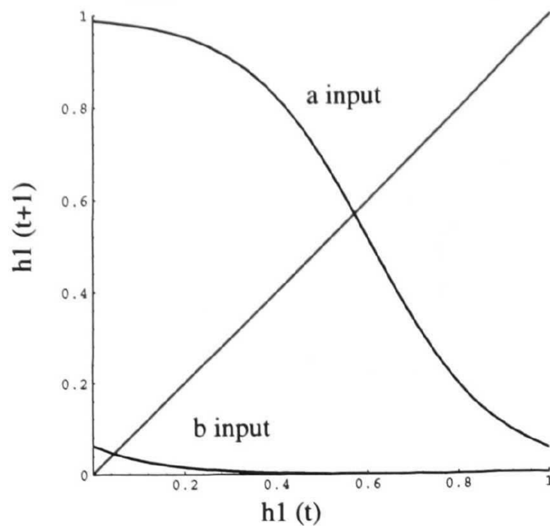


Figure 7: Activation function of hidden unit 1 during presentation of a sequence of *a*s, and *b*s.

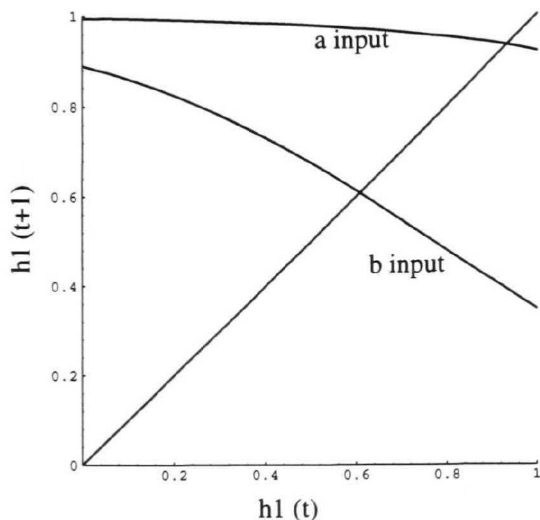


Figure 8: Activation function of hidden unit 2 during presentation of a sequence of *a*s, and *b*s

which corresponds to the number of *a* inputs.

Discussion

We began by posing the question of how a recurrent network might solve a task (i.e., the language $a^n b^n$)which, given a DFA, is known to require a pushdown store. We hypothesized that the network might solve this task by developing a counter.

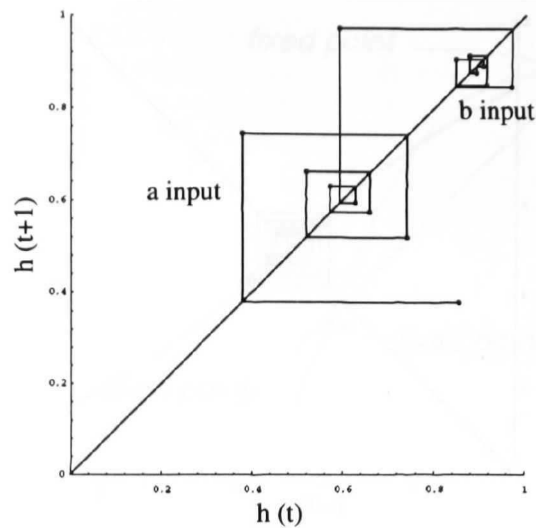


Figure 9: Hidden unit oscillations in trained network, processing 7 *a's* (spiral on lower left, representing hidden unit 1), followed by 7 *b*s (spiral on upper right, representing hidden unit 2).

What we found was something quite different. The solution involved instead the construction of two dynamical regimes. During the first phase, one hidden unit goes into an oscillatory regime in which the activation values converged. We might think of this as akin to the network's "winding up a spring." This phase continues until a *b* is presented. The effect of the *b* is to move the network into the second regime; in this phase the first hidden unit is now damped and the second hidden unit "unwinds" the spring for as long as corresponds to the number of *a*s.

This solution is effective well beyond the depth of strings (n=11) presented during training. Our network was able to generalize easily to length n=21. We found through making additional small adjustments of recurrent weights by hand that the generalization could be extended to n=85.

The solution is interesting because it demonstrates that a task which putatively requires a counter can in fact be solved by a mechanism which shares some but not all the properties of a counter. This particular dynamical solution, for example, solves the problem of indicating when to expect the beginning of a new string; but there is no way to read off from the internal state at any point in time exactly what the current count is (although once in the *b* phase, one can tell how many more *b*s are expected). In this regard, the network is very much like other dynamical systems: The instantaneous view of a child in motion on a swing will not reveal how many times the child has oscillated to get to that position.

We are currently involved in extending this work by looking at related languages such as parenthesis balancing (in which the count may be non-monotonic, as opposed to

the $a^n b^n$ case). We are also interested in cases such as the palindrome language, which more clearly motivate the need for a stack-like mechanisms. Finally, we are developing tools for studying dynamical solutions in networks which have a larger number of hidden units. This poses a major challenge, since the dynamics made possible through the interactions of many hidden units are much more complex than the case studied here.

At this point we prefer not to evaluate this solution as better or worse than that provided by a conventional counter or by the pushdown store of a DFA. We simply note that the solution is different. And we take this as an object lesson that prior notions of how recurrent networks might be expected to solve familiar computational problems are to be regarded as open hypotheses only. We should be prepared for surprises.

Acknowledgments

We are grateful to members of the PDPNLP Research Group at UCSD, and in particular to Paul Mineiros, Gary Cottrell, and Paul Rodriguez for many helpful discussions. This work was supported in part by contract N00014-93-1-0194 from the Office of Naval Research to the second author, and grant DBS 92-09432 from the National Science Foundation, also to the second author.

References

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1, 372.

Elman, J.L. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7, 195-225.

Giles, C.L., Miller, C.B., Chen, D., Chen, H.H., Sun, G.Z., & Lee, Y.C. (1992). Learning and extracting finite state automata with second-order recurrent networks. *Neural Computation*, 2, 331-349.

Kolen, J.F. (1994). Recurrent networks: state machines or iterated function systems? In M. Mozer, P. S. Smolensky, D. Touretzky, J. Elman, & A. Weigend (Eds.), *Proceedings of the 1993 Connectionist Models Summer School* (pp 201-210). Boulder, CO: Lawrence Erlbaum Assoc.

Manolios, P. & Fanelli, R. (1994). First order recurrent neural networks and deterministic finite state automata. *Neural Computation*, 6, 1154-1172.

Pollack, J.B. (1991). The induction of dynamical recognizers. *Machine Learning*, 7, 227.

Seligmann, H.T., & Sontag, E.D. (1992). Neural networks with real weights: Analog computational complexity. Report SYCON-92-05. Rutgers Center for Systems and Control, Rutgers University.

Watrous, R.J., & Kuhn, G.M. (1992). Induction of finite-state languages using second-order recurrent networks. *Neural Computation, 4, 406-414.*