Applied Intelligence 19, 51–63, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

Learning the Dynamics of Embedded Clauses

MIKAEL BODÉN
School of Information Technology and Electrical Engineering, University of Queensland, 14072, Australia
mikael.itee@uq.edu.au

ALAN BLAIR
School of Computer Science and Engineering, University of New South Wales, 2052, Australia
blair@cse.unsw.edu.au

Abstract. Recent work by Siegelmann has shown that the computational power of recurrent neural networks matches that of Turing Machines. One important implication is that complex language classes (infinite languages with embedded clauses) can be represented in neural networks. Proofs are based on a fractal encoding of states to simulate the memory and operations of stacks.

In the present work, it is shown that similar stack-like dynamics can be learned in recurrent neural networks from simple sequence prediction tasks. Two main types of network solutions are found and described qualitatively as dynamical systems: damped oscillation and entangled spiraling around fixed points. The potential and limitations of each solution type are established in terms of generalization on two different context-free languages. Both solution types constitute novel stack implementations—generally in line with Siegelmann's theoretical work—which supply insights into how embedded structures of languages can be handled in analog hardware.

Keywords: recurrent neural network, dynamical system, stack, learning, context-free language, analysis, computation

1. Introduction

Chomsky argued that language competence includes processing of embedded clauses [1]. Moreover, Chomsky pointed out that finite state machines were inadequate for processing languages with embedded clauses. Chomsky doubted that even phrase-structure grammars (which have limited capacity for recursion) would be sufficient to process natural language—implying that even more powerful mechanisms would be necessary (e.g. Turing Machines). Since embedding is an important feature of natural languages, it is natural to ask whether, and in what ways, neural networks might learn the dynamics necessary for processing languages with embedded structure.

It has been proven theoretically that a major class of recurrent neural networks (RNNs) are at least as powerful computationally as Turing Machines [2, 3]. The arguments are based on the analog nature of internal states and network weights (assuming infinite precision). To accommodate recursive computation, where Turing Machines would resort to conventional stacks (or an infinite tape), state spaces are divided (recursively) into smaller state intervals (Siegelmann suggests the use of Cantor sets). Similar approaches have been used to process context-free and context-sensitive languages [4, 5]. However, whether such stack mechanisms can be *learned* by RNNs is a rather different problem. This paper examines RNNs trained on language prediction tasks, to investigate the means by which center embedded language constructs are handled[1] and to see if RNNs exploit continuous state spaces in ways similar to those suggested by Siegelmann's work.

A simple task that requires a stack is predicting the language $a^n b^n$. A valid string from the language consists of a number of *a*s, followed by exactly the same number of *b*s (e.g. *aaabbb*, *ab* and *aaaaaaaabbbbbbbb*). An RNN can learn and generalize to predict the correct number of *b*s when all *a*s have been presented [6]. Briefly put, a trained network counts by "pushing" *a*s and then (whilst *b* is presented) "popping" the stack until it is empty, signalling the end of the string (ready for the next *a*). The dynamics of such networks is usually based on oscillation around fixed points in state space.

Previous work on learning formal languages requiring stacks (e.g. context-free languages) has primarily focused on the use of Simple Recurrent Networks (SRNs) [6–11]. For the $a^n b^n$ prediction task, two main types of solution have been found [7, 9] of which only one generalizes well [10]. This paper reports on experiments along the same lines as previous research but with a second-order recurrent neural network, the Sequential Cascaded Network (SCN; [12]). The SCN frequently finds an additional and novel type of dynamical behavior for predicting $a^n b^n$ which also exhibits good generalization, and whose dynamics we describe in detail.

The paper also investigates how initial training on $a^n b^n$ engenders the dynamical behavior necessary for the more complicated balanced parentheses language (BPL) which, in contrast to $a^n b^n$, allows *a* to re-appear after the first *b* (e.g. *aabaaabbabbb*, *ab* and *aababb*). To illustrate the transfer between the two languages, two sets of simulations are reported, using random initial weights and initial weights taken from successful $a^n b^n$ solutions. In addition to identifying the conditions for successful processing, our study is concerned with the generalization capabilities of the counting behaviors as observed for the two prediction tasks. To characterize each solution type and its potential, the general framework of dynamical systems is used.

2. Background

Recurrent neural networks have been portrayed as offering novel computational features compared with automata and Turing Machines [3, 13]. Their representational power does not stem from an infinite tape (as for Turing Machines) but instead from infinite numerical precision of machine states. Siegelmann and Sontag [2] outline a specific neural network setup for a 2-stack machine equivalent to a 1-tape Turing Machine. Each stack can be manipulated by popping the top element, or by pushing a new element on the stack's top (either 0 or 1). The elements are encoded as a binary stream, $s = s_1, s_2, \ldots, s_n$, into a value

$$z = \sum_{i=1}^{n} \frac{2s_i + 1}{4^i}. \tag{1}$$

This function generates values in the range between 0 and 1 (as is common for sigmoidal outputs) but with a discontinuous and fractal distribution (a Cantor set). The main feature of the function is that for each additional element pushed on the stack, finer precision (larger denominator when expressed as rational values) is required. For example, in a stack which only accepts the element 0 (sufficient to handle both $a^n b^n$ and BPL) the string '00' would generate $\frac{1}{4} + \frac{1}{16}$, while '000' would generate $\frac{1}{4} + \frac{1}{16} + \frac{1}{32}$. Even in the limit, with a continuous flow of zeros, z never reaches $\frac{1}{3}$. The current stack value is encoded by a sigmoidal processing unit and the numerical operations necessary to perform reading, popping and pushing on the stack are easily incorporated as additional units, forming an appropriately weighted neural network [2].

Siegelmann and Sontag [2] show that the encoding strategy extends to n-stacks and can be used to build a Universal Turing Machine. Siegelmann's result, that Turing equivalence can be achieved with rational weights and that "super-Turing computation" can be achieved with real weights, is not limited to the above encoding function. Other functions with similar fractal properties can replace it.

3. Dynamical systems

A discrete time recurrent neural network can usefully be characterized as a discrete time dynamical system [9, 14, 15] with inputs, outputs and state variables.

For both $a^n b^n$ and BPL there are two possible inputs, which define two (non-linear) automorphisms F_a and F_b on the hidden unit activation space. Since these automorphisms can be applied in any order (determined by the input string) it is useful to think of them as comprising an Iterated Function System [16].[2] For a linear autonomous dynamical system on an n-dimensional state space $X \to FX$ where $X \in \Re^n$, an *eigenvalue* of F is a scalar λ and an *eigenvector* (belonging to λ) of F is a vector $\vec{v} \neq 0$ such that $F\vec{v} = \lambda\vec{v}$. The absolute value of the eigenvalue gives the rate of contraction or expansion of F along the principal axis defined by the corresponding eigenvector [9].

In our case the maps F_a and F_b are nonlinear. However, they can be approximated by a linear function in the neighborhood of a fixed point—i.e. a point X in state space for which $X = FX$ (such points can be found by solving the roots of the system of equations for each system). When studied in the vicinity of a fixed point of the system, the Jacobian of the state units has eigenvalues and eigenvectors that express how the nonlinear system changes over time in a neighborhood of that point [9, 17].

When an eigenvalue is positive, 1-periodic (monotonic) behavior occurs along the axis described by the associated eigenvector; when an eigenvalue is negative, 2-periodic (oscillating) behavior occurs. When the (absolute) eigenvalue is below 1 the fixed point is an attractor along the axis given by the eigenvector; when the absolute eigenvalue is above 1 it is a repeller. If a fixed point is attracting along one axis and repelling along another it is also called a saddle point. In some cases eigenvalues take on complex values, thereby indicating a rotation around the fixed point [17]. In the following, the state spaces are 2-dimensional, which means that each fixed point is associated with two eigenvalues and their corresponding eigenvectors.

4. Experiments

4.1. Network Architecture

The architecture we employ is a Sequential Cascaded Network (SCN) with 2 inputs, 2 outputs and 2 state units. The network is equipped with biases in accordance with Pollack's [12] simulations. The output and state activations, at time t, are defined by

$$O_i^t = f\left(\sum_{j,k}^{N_Z,N_I} W_{ijk} Z_j^{t-1} I_k^t + \sum_j^{N_Z} W_{ij\theta} Z_j^{t-1} + \sum_k^{N_I} W_{i\theta k} I_k^t + W_{i\theta\theta}\right) \quad (2)$$

$$Z_i^t = f\left(\sum_{j,k}^{N_Z,N_I} V_{ijk} Z_j^{t-1} I_k^t + \sum_j^{N_Z} V_{ij\theta} Z_j^{t-1} + \sum_k^{N_I} V_{i\theta k} I_k^t + V_{i\theta\theta}\right) \quad (3)$$

where I is the externally set input vector and Z is a vector of internal *state* units (see Fig. 1). W is the set of weights connecting the products between the input

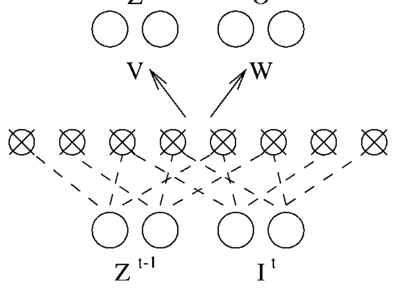

Figure 1. The SCN architecture. Unfilled circles correspond to units in the network. The output, O^t, is determined by first performing a modified outer product between the current input I^t and the state from the previous timestep, Z^{t-1}, multiplied as indicated by the dashed lines. The resulting elements are illustrated using crossed circles. The outer product is then fed through a fully connected weight layer denoted by V and W (illustrated by the arrows). Biases of the weight layer are included in the weight matrices.

and the previous state with the output units. V similarly governs the activation from the input and the previous state to the state layer. N_Z is the number of state units and N_I is the number of input units. Biases are introduced as additional elements in the weight matrices (indexed by θ). Thus, as an example of notation, W_{ijk} is a second order weight (used for connecting the product between the jth state unit and the kth input, with the the ith output unit), $W_{ij\theta}$ and $W_{i\theta k}$ are first order weights (feeding the ith output with either the jth state unit or the kth input), and $W_{i\theta\theta}$ is a first order bias (associated with the ith output unit). The logistic transfer function, $f(x) = 1/(1 + e^{-x})$, is used.

The SRN (which is used as a comparison) is defined by

$$O_i^t = f\left(\sum_j^{N_Z} W_{ij} Z_j^t + W_{i\theta}\right) \quad (4)$$

$$Z_j^t = f\left(\sum_k^{N_I} V_{jk} I_k^t + \sum_i^{N_Z} Z_i^{t-1} U_{ji} + V_{j\theta}\right) \quad (5)$$

where V is the set of weight connecting the input with the state units, W is the set of weights from the state units to the output weights, and U is the set of recurrent weights connecting the state units with themselves (see Fig. 2).

4.2. The $a^n b^n$ Prediction Task

The network is trained, using backpropagation through time (BPTT; [18]) to predict the next letter in

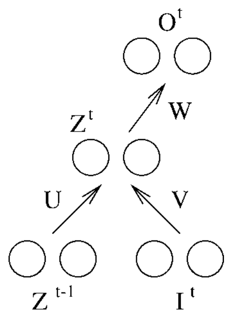

Figure 2. The SRN architecture. Circles correspond to units in the network. The output, O^t, is determined from the current state, Z^t, and the output weights denoted by W. The state is calculated from the previous state, Z^{t-1}, and the current input, I^t. U denotes the recurrent, temporally delayed weights and V denotes the input weights. Biases of the weight layers are included in the weight matrices.

consecutively presented strings of the form a^nb^n for $1 \leq n \leq 10$ (in random order). The legal strings can be described by simple recursive rules:

$$R_1\colon S \rightarrow aXb$$
$$R_2\colon X \rightarrow nil$$
$$R_3\colon X \rightarrow aXb$$

Strings from this language were generated randomly by first applying R_1, then applying R_2 and R_3 randomly according to a probability distribution, $P(R_2) + P(R_3) = 1$. A variable called *level* is used to keep track of the number of levels that the current string has, i.e. how many times R_3 has been applied (plus 1). When $level = 1$, $P(R_2)$ is low and $P(R_3)$ is high. $P(R_2)$ increases monotonically until $level = 10$ when $P(R_2) = 1$. In order to increase the proportion of shorter strings (which is known to enhance learning in some cases [10, 19]) the probability distribution of the presented tokens is skewed so that $P(R_2) > P(R_3)$ when $level > 3$.

Interestingly, the limited resources of the network encourage it to develop dynamics as if the language were unbounded (any n), and it is often able to correctly predict strings a^nb^n for values of n larger than those in the training set. Note that the prediction task is basically only deterministic during the presentation of b since n varies randomly. Having processed all as the network is required to correctly predict b until the final letter of the current string is presented and then predict a (the first letter of the next string). The learning task is considered difficult and requires the network to undergo several fundamental changes in its dynamical behavior (bifurcations) [10].

The symbol a is represented as (1, 0) and b as (0, 1) at the input and output layers of the network. To qualify as a correct output it needs to be on the right side of the output decision boundaries (defined by the output vector (0.5, 0.5)). The criterion for a successful network (a solution) is that it is able to process all strings a^nb^n for $1 \leq n \leq 10$. As can be seen from Eq. (2), the output O^t depends not only on Z^{t-1} but also on the input I^t. In our case there are two possible inputs, so two decision boundaries in Z^{t-1} can be determined and plotted of which only one is effective at a time.

A population of networks, with various initial weights, learning rates, momentum terms and levels of unfolding for BPTT, were trained on a maximum of 10000 strings from a^nb^n. This is typically (validated through numerous simulations) sufficient for convergence. If a network was successful at any time during the 10000 strings, its weights were saved for further analysis. Around 150 of 1200 networks were deemed successful. The best configuration used a learning rate of 0.5, momentum 0.0 and unfolded the network 10 time steps, and generated 40/200 solutions. The fixed points were determined for each successful network, and associated eigenvalues and eigenvectors were calculated.

There are two basic ways SRNs implement the counting necessary for a^nb^n [9]: oscillatory and monotonic. The *monotonic* solution employs gradual state changes in one general direction for each presented a relative to an attractive fixed point and then corresponding state changes in the opposite direction for each presented b relative to a different fixed point. With the SCN the monotonic solution was found in about 5% of the runs, and only when the level of unfolding was kept low (3 timesteps). The monotonic solution type generalizes poorly and is excluded from the analysis below. According to the simulations, SCNs employ both the oscillatory and the monotonic solution type, but also an additional type, entangled spiraling. We treat each of the remaining two solution types in turn.

4.3. Counting by Oscillation

Rodriguez et al. [9] analysed two different SRN weight-sets, which were oscillating, generated by BPTT on training data from a^nb^n, and identified conditions for the SRN to represent and predict a^nb^n correctly to a level beyond the training data. The particular solution type that Rodriguez et al. [9] focused on has one main fixed point for each system. The fixed point for F_a attracts along both axes but with a double periodicity in (at least) one of them. The fixed point of F_b is a saddle point, attracting along one axis and repelling (2-periodically) along the other. The solution type was previously discovered by Wiles and Elman [6] and the experiments herein show that the same basic behavior occurs in the SCN.

The process can be described in terms of the activation of the state units. The state first oscillates, for each a presented to the network, toward the fixed point of F_a. When the first b is presented to the network, the state shifts to another part of the state space close to the F_b saddle point and starts oscillating away from the saddle point until it crosses the output decision boundary, signalling that the next letter is an a (see Fig. 3 for an SCN solution).

The first condition of [9] is that the largest absolute eigenvalues of the two systems should be inversely proportional to one another (in practice one of them is usually around -0.7, the other around -1.4). The inverse proportionality ensures that the rate of contraction around the fixed point of F_a matches the rate of expansion around the saddle point of F_b ([9], p. 23).

The second condition is that the fixed point of F_a should lie on the axis given by the eigenvector corresponding to the smallest absolute eigenvalue of F_b (the direction in which the saddle point attracts) when projected through the F_b fixed point ([9], p. 23). This configuration basically entails the first thing that happens in F_b is a long-distance state shift along its eigenvector to a part in state space close to the fixed point of F_b. The positioning of the eigenvector ensures that the final state of F_a (which identifies the a-count) correctly sets the starting point for the expansion in F_b. The small eigenvalue ensures a direct transition.

About 80% of the successful networks employed counting by oscillation. A sample of 60 networks was used to verify the above criteria. The average largest absolute eigenvalue of F_a is $\lambda_a = -0.68$ ($SD = 0.07$). The average largest absolute eigenvalue of F_b is $\lambda_b = -1.42$ ($SD = 0.10$). The average product of these two is $\lambda_a \lambda_b = 0.96$ ($SD = 0.07$), clearly conforming to the first condition.

The average distance between the eigenvector of the smallest (absolute) eigenvalue of F_b, when projected through the F_b fixed point, and the fixed point of F_a is 0.00 ($SD = 0.00$). Thus, the second condition is confirmed.

There is also an informal criterion which states that the ending point for F_b is a good starting point for F_a ([9], p. 23). However, it does not make sense to measure this on networks which are already deemed successful.

4.4. *Counting by Spiraling*

In the case of the (second order) SCN networks, a new type of solution frequently arose, which had not been observed for the (first order) SRN networks. This type of solution involves a *rotation* towards the F_a attracting fixed point while a is presented, and rotation away from the F_b fixed point (close to the F_a fixed point) in the opposite direction while b is presented, until the decision boundary is reached signalling that a is predicted (see Fig. 4).

All entangled spirals have complex eigenvalues for the Jacobians which indicate that the behavior along the eigenvectors is rotational. A complex number, $x + yi$, can be written on polar form, $r(\cos\theta + i\sin\theta)$, where θ is the angle (in radians) of the complex number perceived as a vector of length r. The rotation is counterclockwise (if $y > 0$) through θ radians followed by an expansion or contraction along the eigenvector by a factor of r [20]. Every complex eigenvalue must be accompanied by its complex conjugate. Apart from the eigenvalues being complex, we can identify a number of informal conditions which contribute to a successful solution.

The first condition is that the fixed points of F_a and F_b should be close to each other, to ensure that the winding up of F_a is correctly balanced by the unwinding of F_b.

The second condition is that the F_a fixed point be attractive, indicated by an absolute eigenvalue (r_a) below 1, or possibly slightly above 1. When the absolute value is exactly 1 the system undergoes a so-called Hopf-bifurcation whereby an invariant circle around the fixed point is born [17]. When the absolute eigenvalue is above 1 the invariant circle forms a boundary (hereafter referred to as the H-boundary) between those states which converge to it from the inside and those that converge to it from the outside. The fixed point is repelling whenever the absolute eigenvalue is above 1 but for those states that are outside the circle boundary the fixed point can be perceived as being attractive. If the absolute value is just above 1 the H-boundary is positioned close to the fixed point enabling F_a to utilize the fixed point in a similar fashion as for a genuinely attractive fixed point.

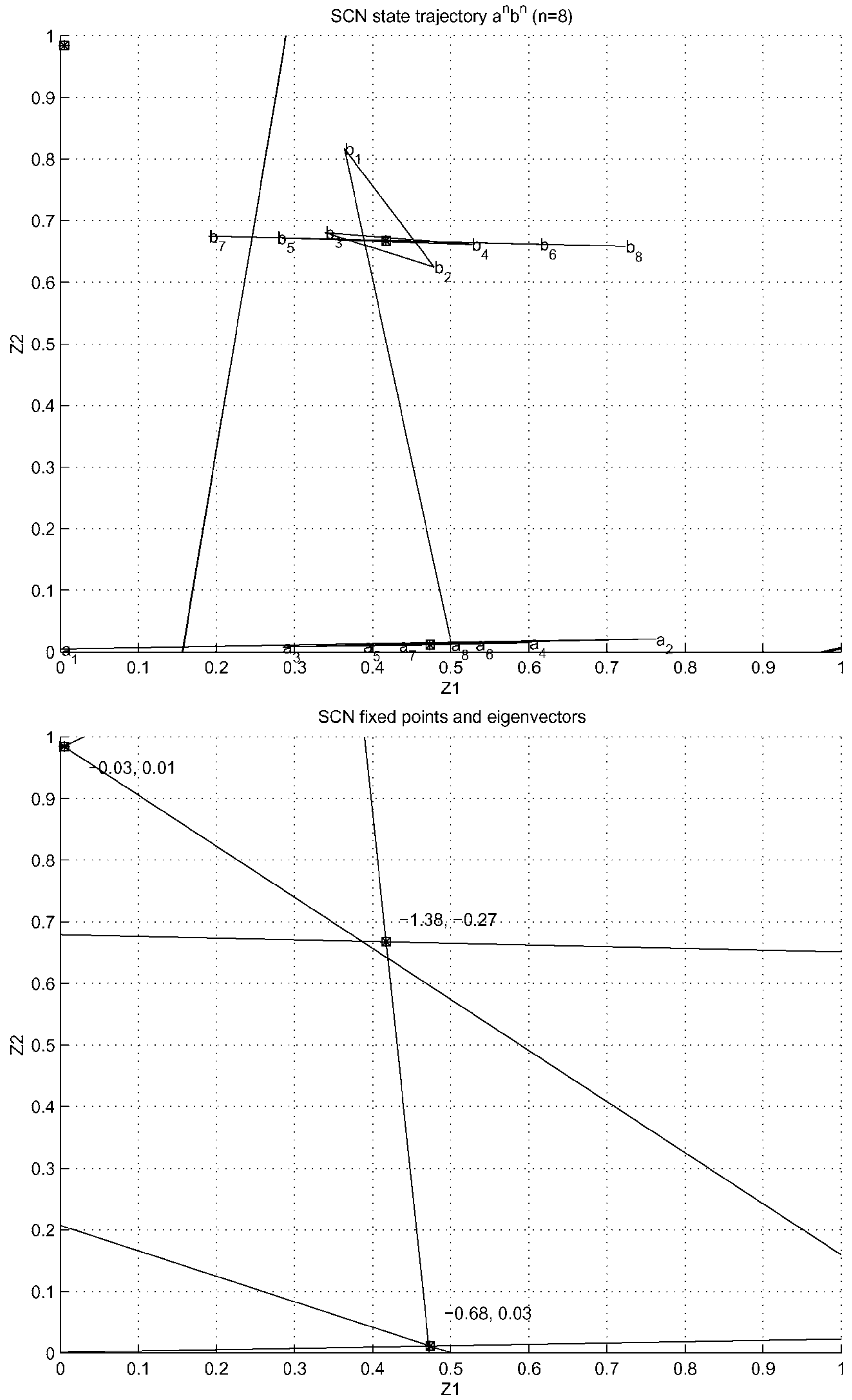

Figure 3. Top: A state trajectory when $a^8 b^8$ is presented to the network. A decision boundary for b inputs is visible and distinguishes between those states that predict b and those that predict a. The decision boundary is based on the weights utilized as a result of the outer product involving (0, 1) (the b input). Note that the a decision boundary is not visible and that, in contrast with the SRN, the output in state space is delayed with one time step (see Eqs. (2) and (4)). Hence, the visible output boundary separates the 6th from the 7th b. Bottom: The fixed points and the associated eigenvectors (projected through the corresponding fixed points) and eigenvalues.

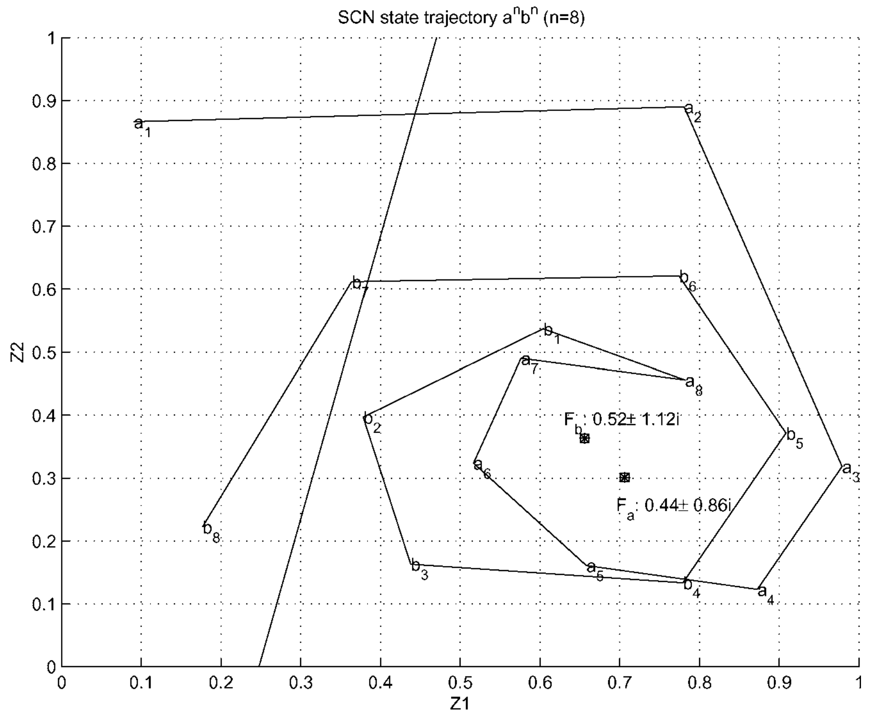

Figure 4. A state trajectory for a^8b^8 and the fixed points for F_a and F_b. Note that the a decision boundary is not visible and that, in contrast with the SRN, the output in state space is delayed with one time step (see Eq. (2)). Hence, the visible output boundary separates the 6th from the 7th b.

The third condition is that r_b, the absolute eigenvalue for F_b, should be large enough to accommodate the full size of the spiral. The F_b fixed point has a larger absolute eigenvalue than the F_a fixed point, positioning the H-boundary further outwards relative to itself. The starting point for F_b is close to the fixed point (see the first condition) and the states diverge until they approach the H-boundary (if it is within the state space). r_b effectively imposes a strict limit on the number of pop-operations before the spiral merges with the H-boundary.

The fourth condition is that $\theta_a \approx \theta_b$ (the angles for the eigenvalues of F_a and F_b, respectively)—to ensure that the rotations of F_a and F_b are synchronized.

The spiraling solution was employed by about 15% of the successful networks but only when the level of unfolding was kept high. When the network was unfolded 10 timesteps, approximately 30% of the successful weightsets employed the entangled spiral. 20 networks were subject to further analysis. The average distance between fixed points of F_a and F_b is 0.05 ($SD = 0.03$), ensuring that the end state for F_a is close to the ideal starting point of F_b.

The average eigenvalues of the F_a fixed point are $\lambda_a = 0.48 \pm 0.83i$ ($SD = 0.18$), and for the F_b fixed point the averages are $\lambda_b = 0.61 \pm 1.08i$ ($SD = 0.28$). The second and third conditions state that the absolute values of the F_a and F_b fixed points should be slightly below 1 and above 1, respectively. The averages are $r_a = 0.97$ ($SD = 0.08$) and $r_b = 1.27$ ($SD = 0.12$).

The average difference between angles (in radians, absolute values) is 0.03 ($SD = 0.03$), verifying the fourth condition. Other general observations include that the real part of the F_a eigenvalue is always slightly smaller than that of F_b. Similarly, the imaginary part of the F_a eigenvalue is, in all cases, slightly smaller than that of F_b.

4.5. Generalization Ability

To investigate generalization ability of the solution types, each successful network was subjected to further training. Each network was trained for a maximum of 10000 strings ($1 \leq n \leq 10$) with a small learning rate, $\eta = 0.1$.[3] The generalization ability was tested continuously and the best performance for each network was recorded.

For the oscillatory solution the generalization was, on average, up to $n = 18.9$ ($SD = 5.9$), and maximum $n = 29$. The spiraling solution achieved, on average, up to $n = 12.6$ ($SD = 3.0$), and maximum $n = 19$.

4.6. SRN and SCN

It is interesting to note that the spiraling solution type is often found by the SCN, but is never found by an SRN with the same number of hidden units. The main reason for this seems to be that in the case of the SRN, the maps F_a and F_b are determined by the same set of weights (excluding bias connections) making it impossible for them to implement spiraling in opposite directions. In contrast, the multiplicative weights of the SCN allow the two maps to vary independently.

Additionally, the SRN requires that the hidden-to-output connections define "predict a" and "predict b" regions in the hidden unit activation space which are independent of the current input; the input-to-output connections of the SCN allow these regions to depend also on the current input, affording more flexibility to the SCN. A similar advantage of SCN is noted for learning a simple context-sensitive language with fewer state units than the SRN [21].

5. Balanced Parentheses Language

Similar to strings from $a^n b^n$, a BPL string, e.g. *aaaabbabbb*, requires that the network pushes as and that it pops the stack whilst b is presented. Due to the random generation of strings from the language, it is not possible for the network to correctly predict all bs since another a may turn up at any time and not only when a new string is started. The language can be described by simple recursive rules:

$$R_1\colon S \to aX^*b$$
$$R_2\colon X \to nil$$
$$R_3\colon X \to aX^*b$$

Similar to the $a^n b^n$ generator, there is a variable *level* which indicates the current R_3 count (plus 1). The difference compared with $a^n b^n$ is that R_1 and R_3 allow a selection of any number of intervening Xs, symbolized by X^*.[4] Consequently, by traversing paths created by multiple substrings, the *level* can go up, down and up again.

As with $a^n b^n$, each string from BPL is generated by first applying R_1 and then applying R_2 and R_3 interchangeably according to a probability distribution. There are essentially two phases involved. In phase 1, $P(R_2)$ is low (and $P(R_3)$ is high), while in phase 2 the opposite is true. At any time during phase 1 there is a $1/3$ chance that the generator will enter phase 2. When *level* reaches 10 the generator is forced to switch into phase 2 if it has not already done so. When R_1 or R_3 is employed the generator also picks a number of substrings to generate with a constant probability. The probability for generating 1 is much higher than for generating 2 and decreases rapidly for higher numbers. As a whole, the above strategy enforces preference to short and $a^n b^n$-like strings.

Two data sets were used in the experiments. The training set was generated according to a probability distribution with $P(R_3) = 0.95$ for phase 1 and $P(R_3) = 0.05$ for phase 2, which generates strings from $a^n b^n$ to a high degree. The test set was generated using $P(R_3) = 0.80$ for phase 1 and $P(R_3) = 0.20$ for phase 2, which contains more complicated strings with frequent intervening substrings (more bs are introduced earlier in the string, and more as appearing after the first b).

Rodriguez et al. [9] showed how the SRN can maintain the coordination of trajectories such that for substrings (multiple Xs in R_1 or R_3) the network returns to a state still allowing a correct "end-of-string" prediction (predict the first letter of the next string). The additional condition for the oscillatory solution is that the fixed point of F_b should lie on the eigenvector associated with the smaller absolute eigenvalue of F_a when projected through the F_a fixed point. Thereby, the two systems can exchange "counting information" at any time, by crossing the state space along the appropriately aligned eigenvectors of both F_a and F_b. Our SCNs demonstrate the same characteristics (see Fig. 5). Notably, the same strategy effectively incorporates a recognition task for BPL strings. The decision boundary for signalling that an a is due (during the presentation of bs) can be used to mark that the presented string is grammatically correct.

In the entangled spiral there is no explicit need to exchange counting information since states of F_a and F_b are largely superposed. For example, the state arrived at after aa has been presented is approximately the same

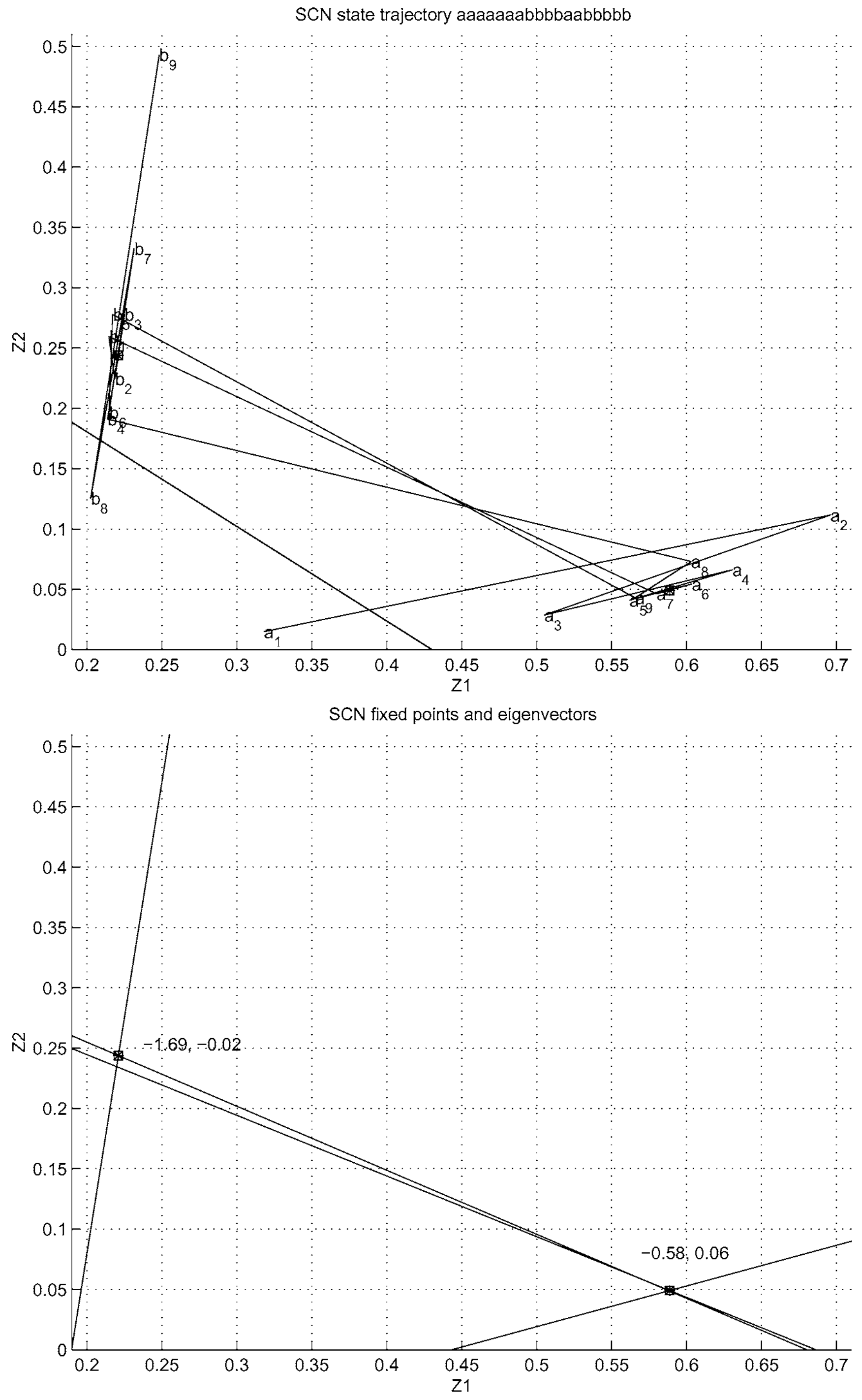

Figure 5. A state trajectory for *aaaaaaabbbbaabbbbb* (above). The fixed points and the associated eigenvectors and eigenvalues (below).

as after *aaaabb* has been presented (same *level* in both cases). The interpretation of the state is independent of which system is currently active. Hence, there is no additional constraint, only a stronger enforcement of the existing conditions that the dilation of F_b matches the contraction of F_a ($\theta_a = \theta_b$) with great precision and that the fixed points of F_a and F_b are close in state space (see Fig. 6 compared with Fig. 4). Thus, spirals

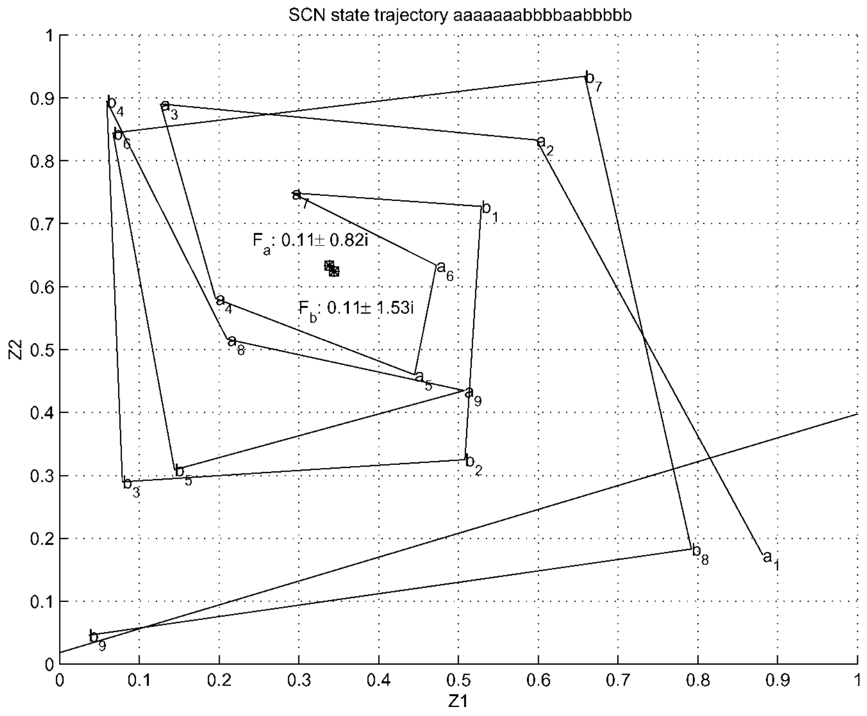

Figure 6. A state trajectory for *aaaaaaabbbbaabbbbb* and the fixed points for F_a and F_b.

with good generalization performance on $a^n b^n$ can be expected to also manage BPL.

Using the criterion that a successfully processed string from BPL is one for which the network correctly predicts the first letter of the following string, the test set was used to rank each network's degree of success. 20 networks successfully predicting $a^n b^n$ for each solution type were trained further using BPL for a maximum of 20000 strings with a few different configurations. A learning rate of 0.1 (no momentum) and an unfolding level of 20 turned out to be the best. The best configuration produced 6 (of 20) BPL predictors of the oscillation type, and 5 (of 20) BPL predictors of the entangled spiral type, all completely successful on 20 random strings from the BPL test set. All networks maintained the solution type they produced for the $a^n b^n$ data. To further investigate each network that managed the first test set, another 100 random test strings from BPL were generated. The average score on the second test set was 78 ($SD = 24$, best $= 100/100$) among 10 networks of the oscillation solution type and 52 ($SD = 29$, best $= 91/100$) on 10 networks of the entangled spiral type.

Training set probabilities that entailed more complicated strings rendered worse training performance. 300 networks were trained on BPL from small random initial weights with various learning parameters but no solutions were found according to the criteria described above. Thus, successful learning of BPL requires good initial weights.

6. Discussion

The basic idea behind Siegelmann's proofs on Turing equivalence of certain RNNs is that state spaces of stacks are divided into smaller and smaller intervals of values as elements are pushed. Hence, the need for an infinitely large discrete state space is transformed into a need for a continuous state space with infinite precision. Apart from the apparent transition to two dimensions (the Cantor set only requires one), the two observed learned behaviors recall the basic idea: As elements are pushed onto the stack the state space is recursively divided into smaller regions. Both solution types employ behavior where states oscillate/rotate around a fixed point and push elements on the stack by moving closer and closer to it. Similarly, elements are popped by moving in a similar fashion (and speed) away from another fixed point (see Fig. 7).

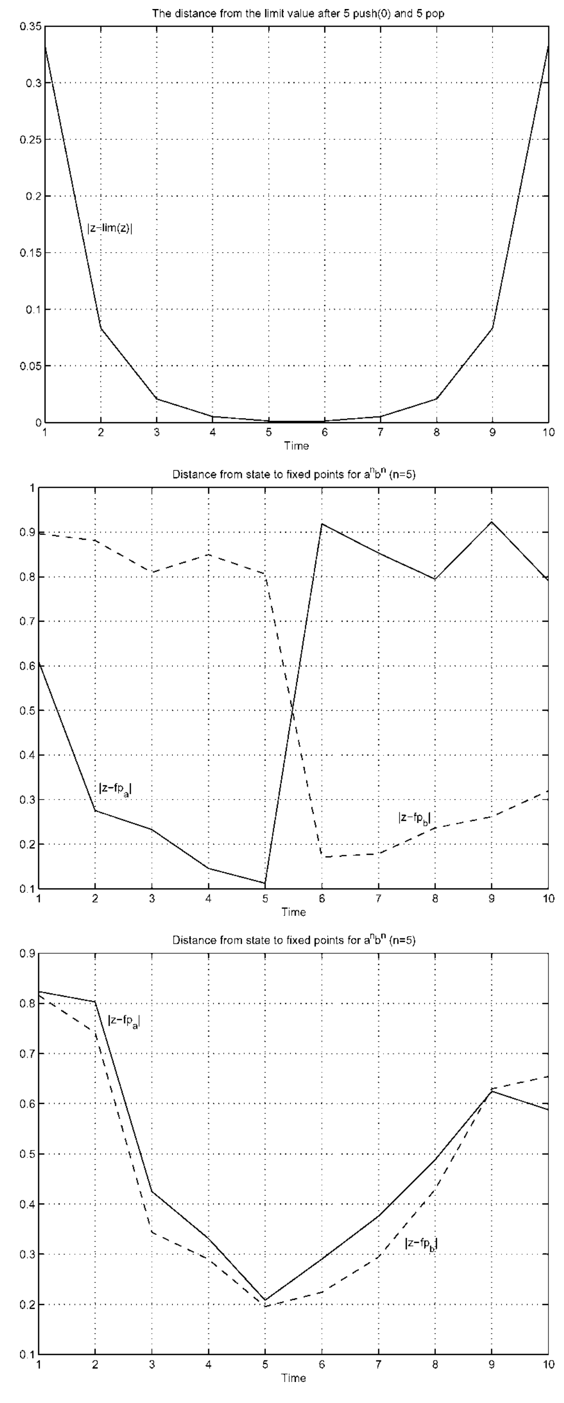

Figure 7. A perspective on the encoding of the string $aaaaabbbbb$ over time. In the first graph the Cantor encoding (see Eq. (1)) is shown as seen from the limit value 1/3. The second graph shows the encoding by a typical oscillating solution and the third graph shows the encoding by a typical entangled spiral solution, both as seen from the two main fixed points (the distance between the state and the F_a fixed point is a solid line, and the distance between the state and the F_b fixed point is a dashed line).

The push and pop operations defined for the Cantor set operate independently of the preceding operations and do not leave a trace. However, the oscillating solution generates states in two parts of state space, distinguishing between states encoded by F_a and those encoded by F_b. Thus, the state for aaa is different from the state for $aaaaabb$. Similar to the Cantor set, the entangled spiral has state trajectories which are aligned so as to minimize any difference between states generated by the two systems.

In general, the learned behavior we observe is different from the Cantor set approach in the sense that the precision is not strictly enforced. Weights are not sufficiently precise to exhibit true fractal nature and enable generalization to infinitely deep levels. However, the performance degradation observed in the trained networks for deep input strings is also apparent in human language performance [11].

7. Conclusion

There are a variety of dynamics available for implementing stack-like behavior in RNNs. The present study has explored two types of dynamics which are (1) learnable, (2) unlike classical automata, only limited by numerical precision and not memory, and (3) conforming to Siegelmann's [2, 3] principal suggestion that stacks can be built in analog hardware by recursively subdividing state spaces.

On the technical side, SCNs employ three counting behaviors, one of which is not found by the SRN with equally many units. Of the three, two have good potential for handling BPL: oscillation and entangled spiral. Both exploit the principle of recursively dividing a region in state space into smaller and smaller pieces as elements are pushed onto the stack and conversely expanding in state space while the elements are popped.

The experiments confirm that networks employing the oscillating solution type handle $a^n b^n$ and BPL according to previously identified criteria [9]. The criteria also hold for a different (higher order) network type. The conditions for successfully employing the entangled spiral dynamics can be described in terms of complex eigenvalues of the Jacobian of the state units.

Moreover, experiments establish that the oscillatory type is found more often by BPTT compared with the entangled spiral type. Further, the oscillation type generalizes better to both $a^n b^n$ and BPL, a fact that clashes with the qualitative observation that entangled spirals more naturally lend themselves to BPL compared with oscillating solutions.

Overall, the present study suggests an alternative explanational framework based on analog recurrent neural networks for illustrating how mechanisms for embedded languages can be learned and implemented. The results are generally in line with Siegelmann's theoretical work and in sharp contrast with classical automata. Moreover, the theoretical work corresponds to what (within e.g. generative linguistics) could be regarded as a language competence theory and the simulation results—which are admittedly not achieving infinite embedding—indicate language performance factors.

Acknowledgments

The authors gratefully acknowledge feedback from Morten Christiansen and two anonymous reviewers on a draft of this paper and discussions with Janet Wiles, Brad Tonkes and Stephan Chalup. The main part of this work was carried out while the first author was at the Department of Computer Science and Electrical Engineering, University of Queensland, and supported by a grant from the Australian Research Council.

Notes

1. It is important to acknowledge that human language users do not tolerate many levels of center embedding. This is sometimes referred to as the *performance* aspect of the language. However, at the core of many linguisitic theories (including Chomsky's generative grammar; see [1]) is language *competence* (in which language is genuinely recursive).
2. Note, however, that many of the convergence proofs for Iterated Function Systems rely on linearity of the automorphisms.
3. Several different learning rates were tested of which 0.1 was close to optimal for both solution types.
4. If only one intervening substring is allowed, $a^n b^n$ is generated.

References

1. N. Chomsky, *Syntactic Structures*, Mouton, The Hague, 1957.
2. H.T. Siegelmann and E.D. Sontag, "On the computational power of neural nets," *Journal of Computer System Sciences*, vol. 50, no. 1, pp. 132–150, 1995.
3. H.T. Siegelmann, *Neural Networks and Analog Computation: Beyond the Turing Limit*, Birkhäuser, 1999.
4. S. Hölldobler, Y. Kalinke, and H. Lehmann, "Designing a counter: Another case study of dynamics and activation landscapes in recurrent networks," in *Proceedings of KI-97: Advances in Artificial Intelligence*, Springer Verlag, 1997, pp. 313–324.
5. M. Steijvers and P. Grünwald, "A recurrent network that performs a context-sensitive prediction task," Technical Report NC-TR-96-035, NeuroCOLT, Royal Holloway, University of London, 1996.
6. J. Wiles and J.L. Elman, "Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks," in *Proceedings of the Seventeenth Annual Meeting of the Cognitive Science Society*, Lawrence Erlbaum, 1995, pp. 482–487.
7. B. Tonkes, A. Blair, and J. Wiles, "Inductive bias in context-free language learning," in *Proceedings of the Ninth Australian Conference on Neural Networks*, 1998, pp. 52–56.
8. P. Rodriguez and J. Wiles, "Recurrent neural networks can learn to implement symbol-sensitive counting," in *Advances in Neural Information Processing Systems*, edited by Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, The MIT Press, 1998, vol. 10.
9. P. Rodriguez, J. Wiles, and J.L. Elman, "A recurrent neural network that learns to count," *Connection Science*, vol. 11, no. 1, pp. 5–40, 1999.
10. M. Bodén, J. Wiles, B. Tonkes, and A. Blair, "Learning to predict a context-free language: Analysis of dynamics in recurrent hidden units," in *Proceedings of the International Conference on Artificial Neural Networks*, Edinburgh, 1999, pp. 359–364. IEE.
11. M. Christiansen and N. Chater, "Toward a connectionist model of recursion in human linguistic performance," *Cognitive Science*, vol. 23, pp. 157–205, 1999.
12. J.B. Pollack, "The induction of dynamical recognizers," *Machine Learning*, vol. 7, p. 227, 1991.
13. C. Moore, "Dynamical recognizers: Real-time language recognition by analog computers," *Theoretical Computer Science*, vol. 201, pp. 99–136, 1998.
14. M. Casey, "The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction," *Neural Computation*, vol. 8, no. 6, pp. 1135–1178, 1996.
15. P. Tino, B.G. Horne, C.L. Giles, and P.C. Collingwood, "Finite state machines and recurrent neural networks—automata and dynamical systems approaches," in *Neural Networks and Pattern Recognition*, edited by J.E. Dayhoff and O. Omidvar, Academic Press, 1998, pp. 171–220.
16. M. Barnsley, *Fractals Everywhere*, Academic Press: Boston, 2nd edition, 1993.
17. R.L. Devaney, *An Introduction to Chaotic Dynamical Systems*, Addison-Wesley, 1989.
18. R.J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.
19. J.L. Elman, "Learning and development in neural networks: The importance of starting small," *Cognition*, vol. 48, pp. 71–99, 1993.
20. M.W. Hirsch and S. Smale, *Differential Equations, Dynamical Systems, and Linear Algebra*, Academic Press: New York, 1974.
21. M. Bodén and J. Wiles, "Context-free and context-sensitive dynamics in recurrent neural networks," *Connection Science*, vol. 12, no. 3, 2000.

Mikael Bodén received the Ph.D. degree in Computer Science from the University of Exeter, UK, in 1997. He has held academic and research positions with the University of Skövde, Sweden, and Halmstad University, Sweden. He is presently with the University of Queensland, Australia (http://www.itee.uq.edu.au/∼mikael). His research interests include recurrent neural networks, machine learning, language, bioinformatics, evolutionary and emergent computation and cognitive science.

Alan Blair received B.Sc. and B.A. degrees from the University of Sydney in 1988/89, and a Ph.D. in mathematics from M.I.T. in 1994. He went on to do research in computer science at the Universities of Brandeis, Queensland and Melbourne before taking up his current position at the University of New South Wales.

His research interests include machine learning, evolutionary and emergent computation, robotics, recurrent neural networks and language processing.