Constructive Learning of Recurrent Neural Networks: Limitations of Recurrent Casade Correlation and a Simple Solution *

C.L. Giles[a,b], D. Chen[a], G.Z. Sun[a], H.H. Chen[a], Y.C. Lee[a], M.W. Goudreau[c]

[a]Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

[b]NEC Research Institute
4 Independence Way
Princeton, NJ 08540

[c]Dept. of Computer Science
University of Central Florida
Orlando, FL 32816

Abstract

It is often difficult to predict the optimal neural network size for a particular application. Constructive or destructive methods that add or subtract neurons, layers, connections, etc. might offer a solution to this problem. We prove that one method, Recurrent Cascade Correlation, due to its topology, has fundamental limitations in representation and thus in its learning capabilities. It cannot represent with monotone (i.e. sigmoid) and hard-threshold activation functions certain finite state automata. We give a "preliminary" approach on how to get around these limitations by devising a simple constructive training method that adds neurons during training while still preserving the powerful fully-recurrent structure. We illustrate this approach by simulations which learn many examples of regular grammars that the Recurrent Cascade Correlation method is unable to learn.

1 Introduction

Choosing the architecture of a neural network for a particular problem usually requires some prior knowledge of the problem's complexity and/or usually involves a lot of trial-and-error. However, the network topology directly affects the two most important factors of neural network training, *e.g.* generalization and training

*Published in *IEEE Transactions on Neural Networks* vol.6, no. 4, p. 829, 1995.

time. Both theoretical studies [38] and simulations [27] show that larger than necessary networks tend to overfit the training data and thus have poor generalization; while too small a network will have difficulty learning the training samples. In general a large network will also require more computation than a smaller one. Similar to directions previously explored (see references below), we treat the topology of network as a trainable parameter and allow the network to adjust its structure according to the problem to be solved.

Currently, there are no formal methods to customize or select the "right" network structure. What follows is a brief and illustrative discussion of some of the many methods discussed in the literature. This list is not meant to be inclusive; only to prepare the reader for the development of our approach.

- A common method is that of "**trial-and-error**" – train networks with different sizes and select the smallest one that learns the samples. This approach usually requires quite a bit of experience in training on the particular problem at hand in order to select the optimal structure.

- Another approach is to use a selection process such as **genetic algorithms** to select the best building blocks for the network([17]). Obviously, this approach has the disadvantage of requiring a great deal of computation - genetic algorithms often require a very large population from which to choose. Also, since the neural network is a distributed system, it is unlikely that a good building block in one network will be a good component for another network.

- Still another approach is a **destructive** or **pruning** method ([29]). This method hopes to increase the network's generalization capability by starting with a fairly large network and then removing the unimportant connection weights or units. Approaches include removing the smallest magnitude or insensitive weights or by adding a penalty term to the energy function to encourage weight decay. A recent approach, Optimal Brain Surgery [18], slightly adjusts the remaining weights after other weights are removed.

- Finally, there are **constructive** or **growth** methods where the network starts with a small size and grows when needed. The obvious advantage of this method is that it might require less computation than the destructive methods. In addition, this method frees one from "guessing" a large enough initial network size and topology. However, without proper control of the growth, this process could lead to an oversized network. It is also possible to combine constructive and destructive methods.

Recurrent Neural Networks have ability to process and store temporal information and sequential signals. For example recent work has shown that various recurrent networks are able to infer small regular grammars from examples [3, 11, 27, 36, 40]. However, one of the problems associated with recurrent networks is that the training scales badly with both network and problem size [41]. The convergence time can be very slow and

training errors are not always guaranteed to reduce to previously defined tolerances. Thus, pure destructive-based methods are at a disadvantage since an oversized network must first be trained. By using constructive training methods, the hope is that the neural network could build itself incrementally and in the process speed up the training process. We show that an existing recurrent network constructive method, Recurrent Cascade Correlation (RCC), has fundamental representational limitations when it comes to representing finite-state processes such as regular grammars. We propose an alternative method which eliminates these limitations.

2 Constructive Learning

A constructive method dynamically grows the network structure during the network's training. Various constructive methods have been studied and various types of network growing methods have been proposed [2, 4, 6, 9, 10, 16, 20, 25, 26, 33, 37]. A good review of some of these algorithms is found in [19]. To our knowledge "all" constructive methods with the exception of recurrent cascade-correlation [7] are for feed-forward networks.

2.1 Recurrent Cascade-Correlation

Recurrent cascade-correlation (RCC) is a "recurrent version" of cascade-correlation (CC). We first review the Cascade-Correlation constructive algorithm (figure 1). To quote Fahlman [6]: "Cascade-Correlation combines two key ideas: the first is the cascade architecture, in which hidden units are added to the network one at a time and do not change after they have been added. The second is the learning algorithm, which creates and installs the new hidden units." (The constructive architectural method of CC is very similar to the pyramid method proposed by Gallant [10].) In CC a preset number of single layer neurons are initially trained until the error is no longer significantly reduced. If at this point the error is not satisfactory, a single hidden unit is added and the previously trained weights are frozen. The added hidden unit connects to the original inputs and all pre-existing hidden units. The trainable weights can be updated using the desired optimization algorithm such as backpropagation; for more details see [6]. The training time for this algorithm appears to be very fast since only the newly added weights are trained.

The "recurrent" extension to cascade-correlation is straightforward [7]. Each added neuron has a time-delayed recurrent self-loop (see figure 2). This gives RCC state memory. However, we will show that this self-loop state memory is restricted in the sense that it is insufficient to represent all finite state automata. Again, all previous weights other than the newly added feedforward and recurrent ones are frozen. Since the representational restrictions of RCC are not due to the training procedure, we do not discuss them here and refer the interested reader to Fahlman [7].

2.2 Criteria for Network Growth

In addition to the normal updating and learning rules of the neural network, a constructive training scheme must also address how to architecturally (or topologically) change the network. We note the following criteria:

1. when the network structure needs to change,
2. how to connect the newly created neurons (and/or weights) to the existing network,
3. how to assign initial values to the newly added connection weights (what knowledge these new weights must have).

In general, the changing of the network's structure would most likely change the error surface dramatically. For efficient training, we propose the following. The first expansion criterion needs to be carefully chosen so that the network structure is relatively stable and that the network's knowledge before each expansion is maximized. RCC addresses this by changing the networks structure when the error function no longer decreases.

The second criterion should be based on the way that the network will increase its classification or representational power. RCC permits only local connections and no connections directly back to early neuron state values. By adding fully-connected neurons, the growing recurrent network has the capability to establish direct state-memory connections to the the earlest states formed during network growth.

For the third criterion, we propose that just as the network is about to grow, the network preserve as much previously acquired knowledge as possible. (Previous work where rules are encoded directly into the recurrent networks have shown that prior knowledge does improve the learning speed [14, 13].) RCC accomplishes this by freezing all previous weight values after a new neuron is added. Another way to do this is to set the new weights to very small values or zero; thus causing the newly added neurons to initially have little or no effect on training.

3 Simple Dynamically-Driven Recurrent Network

We briefly review recurrent networks; for a more thorough discussion see [19, 22]. A simple dynamically-driven recurrent neural network (RNN) consists of three parts: input layer, a simple recurrent layer and output layer [5]. We term the recurrent network "driven" to denote that it responds temporally to inputs. The hidden recurrent layer is activated by both the input neurons and the recurrent layer itself. The output neurons are in general activated by the input and recurrent neurons, or by only the recurrent neurons. The recurrent neurons selectively keep track of the input information history by forming internal "state"

representations. This memory mechanism allows the recurrent network to handle temporal information of arbitrary length. This system process is similar to that of a finite state automata (FSA) in that the network only keeps the "state" information of the last time step [30].

Connections between layers can be first, second, or even higher orders [11, 32, 40]. In general, the updating rule can be written as,

$$\mathbf{S}^{t+1} = \Theta(\mathbf{W}, \mathbf{S}^t, \mathbf{I}^t), \qquad \mathbf{O}^{t+1} = \Omega(\mathbf{U}, \mathbf{S}^t, \mathbf{I}^t).$$

where $\mathbf{I}^t, \mathbf{S}^t, \mathbf{O}^t$ are the typically real-valued input, recurrent and output neurons at time step t and $\mathbf{W}, \mathbf{U}$ are the respective connection weights. A typical learning rule might use gradient descent [42] to adjust the weights $\mathbf{W}, \mathbf{U}$ so as to minimize the error function ,

$$E = E(\,(\mathbf{T}^1, \mathbf{O}^1), (\mathbf{T}^2, \mathbf{O}^2), \cdots, (\mathbf{T}^t, \mathbf{O}^t)\,),$$

where $\mathbf{T}^t$ is the desired output at each time step t. There are many recurrent training algorithms; for a discussion see [19, 22].

4 Limitations of the Recurrent Cascade-Correlation Architecture

Recurrent Cascade-Correlation (RCC) is illustrated in (figure 2). Regardless of the training procedure, it differs from a fully-connected recurrent network in the sense that direct recurrent connections from the newly-added neurons to the old neurons are restricted – *i.e.* nonexistent. Even though this self-recurrent restriction significantly simplifies the training (each neuron is trained sequentially, thus the name cascade), it also restricts the representational power of the network. We will show that this type of structure with a *monotone (eg. sigmoid) updating function* has only limited memory and is **not** capable of representing all finite state automata (FSA) [21]. (In this paper all FSA are deterministic.) We contend that the representation and learning of finite state automata is representative of a class of problems that are of interest in real-world problems, such as VLSI design or speech processing. And that if the representational capabilities of the recurrent network is restricted, then so is the class of problems that the recurrent net can learn.

For two different activation functions, the hard-threshold and a simple monotonically increasing function such as a sigmoid, we will prove the limitations of RCC. However, for other more complicated activation functions such as radial basis functions this result may not hold.

For simplicity, we consider first-order neural networks where the state neurons S_i^t are recursively updated

as

$$S_i^{t+1} = \Theta(\sum_{i,j} W_{i,j} S_j^t + I_j^t). \tag{1}$$

For the derivations below we have the special case of constant input sequences, say all 1's. The activation function of the first neuron S_1 then can be simplified to

$$S_1^{t+1} = \Theta(W_{11} S_1^t + \theta_1),$$

where update function Θ is the activation function (the constant input term I has been dropped for convenience). The activation function for the nth RCC neuron S_n can be written as

$$S_n^{t+1} = \Theta(W_{n1} S_1^{t+1} + W_{n2} S_2^{t+1} + \cdots + W_{nn} S_n^t + \theta_n),$$

or,

$$S_n^{t+1} = \Theta(W_{nn} S_n^t + \Lambda_n^{t+1}), \tag{2}$$

where

$$\Lambda_n^{t+1} = W_{n1} S_1^{t+1} + W_{n2} S_2^{t+1} + \cdots + W_{n(n-1)} S_{n-1}^{t+1} + \theta_n.$$

4.1 Hard-threshold Neurons

We examine all possible cases for a hard-threshold activation function and constant input. We show that all possible hidden neurons outputs are constant or of period 2, i.e. oscillates between two constant values.

Without loss of generality, assume the hard-threshold Θ takes on the value 0 or 1. Recall equation 2,

$$S_n^{t+1} = \Theta(W_{nn} S_n^t + \Lambda_n^{t+1}).$$

If Λ_n^{t+1} is oscillating at period 2, *i.e.*

$$\Lambda_n^{t+1} = \begin{cases} \lambda_1 & \text{if } t = \text{odd} \\ \lambda_2 & \text{if } t = \text{even} \end{cases}$$

then the update graphs for all possible output sequences of eq. 2 are listed below. Either of the two output sequences listed is possible.

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	0	
λ_1	1	0	*00000000*$\cdots$
λ_2	0	0	*or*
λ_2	1	0	*10000000*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	1	
λ_1	1	0	*01010101*$\cdots$
λ_2	0	0	*or*
λ_2	1	0	*10010101*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	0	
λ_1	1	1	*00000000*$\cdots$
λ_2	0	0	*or*
λ_2	1	0	*11000000*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	1	
λ_1	1	1	*01010101*$\cdots$
λ_2	0	0	*or*
λ_2	1	0	*11010101*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	0	
λ_1	1	0	*00101010*$\cdots$
λ_2	0	1	*or*
λ_2	1	0	*10101010*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	1	
λ_1	1	0	*01010101*$\cdots$
λ_2	0	1	*or*
λ_2	1	0	*10101010*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	1	
λ_1	1	1	*01010101*$\cdots$
λ_2	0	1	*or*
λ_2	1	0	*10101010*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	0	
λ_1	1	0	*00000000*$\cdots$
λ_2	0	0	*or*
λ_2	1	1	*10000000*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	0	
λ_1	1	1	*00000000*$\cdots$
λ_2	0	0	*or*
λ_2	1	1	*11111111*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	1	
λ_1	1	1	*01111111*$\cdots$
λ_2	0	0	*or*
λ_2	1	1	*11111111*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	0	
λ_1	1	0	*00101010*$\cdots$
λ_2	0	1	*or*
λ_2	1	1	*10101010*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	1	
λ_1	1	0	*01101010*$\cdots$
λ_2	0	1	*or*
λ_2	1	1	*10101010*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	0	
λ_1	1	1	*00111111*$\cdots$
λ_2	0	1	*or*
λ_2	1	1	*11111111*$\cdots$

Λ	S^t	S^{t+1}	possible sequences
λ_1	0	1	
λ_1	1	1	*11111111*$\cdots$
λ_2	0	1	*or*
λ_2	1	1	*11111111*$\cdots$

The other two possible combinations of S^{t+1} shown in the table

Λ	S^t	S^{t+1}
λ_1	0	0
λ_1	1	1
λ_2	0	1
λ_2	1	0

Λ	S^t	S^{t+1}
λ_1	0	1
λ_1	1	0
λ_2	0	0
λ_2	1	1

cannot be realized because the hard-threshold function is unable to generate XOR. It is evident from the output sequences listed above that the sequence S_n^t will after the 1st or 2nd character be either a constant or oscillate at period 2. Thus, Λ_n^{t+1} and S_n^{t+1} will exhibit the same behavior.

This proves that for hard-threshold neurons RCC is not capable of representing and thus learning some finite state automata, *e.g.* any automata which consists of a labeled digraph component which for the same input has an output with period greater than 2. From above proof, the simplest subcomponent of a FSA

that RCC cannot represent is the 3-cycle digraph shown in figure 4. An example of such a FSA with this subcomponent is the (Tomita-6) automaton [39].

4.2 Sigmoid Neurons

We now prove for a *monotonic activation function* RCC is **not** capable of representing some finite state automata. The limitations of RCC can be shown by studying the network's dynamics. First we show a well known result from nonlinear dynamics. We formally define the following: a point is said to be a fixed point of a function $f(x)$ if $f(x) = x$, and is called a period-n point (n=0,1,2,...) of $f(x)$ if $f^n(x) = x$ and $f^k(x) \neq x$ $k < n$. A fixed point is simply a period-1 point [23].

Assume a and b are two neighboring fixed points of a monotonically increasing function f(x). In (a, b), either $f(x)$ is always $> x$ or $f(x)$ is always $< x$. Within (a, b), the trajectory of $x^{n+1} = f(x^n)$ will always be in one direction. For a monotonically increasing function, $(f(x)-f(a))/(x-a) > 0$ and $(f(x)-f(b))/(x-b) > 0$, *i.e.*, the trajectory starting in (a, b) will stay in (a, b). Therefore, for a monotonically increasing function f(x), there is no period-n point for $n \geq 2$..

Recall equation 2 for the first neuron S_1. For $W_{11} > 0$ and because of the property above, S_1 will eventually settle to a fixed point, i.e. remain at a constant value (figure 5a). For $W_{11} < 0$, S_1 can also settle to a period-2 point, i.e. oscillate between two constant values. This is because $\Theta(\Theta(.))$ is a monotonically increasing function, figure 5b.

We prove this by induction. Assume that at each time step, $S_1, \cdots, S_{n-1}$ either oscillate with period 2 or remain constant. Since Λ_n^{t+1} is a function of only $S_1, \cdots, S_{n-1}$, it will at most oscillate at period 2. If Λ_n^{t+1} remains constant for all time, then S_n will be a constant or oscillate at period 2. (The reasoning is similar to that for S_1 above.) If Λ_n^{t+1} oscillates at period 2, say between two values λ_1 and λ_2, the activation function for equation 2 can be rewritten as,

$$\begin{aligned} S_n^{t+1} &= \Theta(\lambda_1 + W_{nn} S_n^t), \\ S_n^{t+2} &= f(S_n^t) = \Theta(\lambda_2 + W_{nn}\Theta(\lambda_1 + W_{nn} S_n^t)). \end{aligned}$$

Function $f(S_n^t)$ is monotonically increasing and, therefore, has no period-n points, regardless of W_{nn}'s value (figure 6). This means $S^0, S^2, S^4, \cdots, S^t, S^{t+2}, \cdots$ will approach a constant value, or will at most oscillate with period 2.

Thus, the RCC network structure with a monotone activation function will at some point fail to correctly classify constant input sequences that generate periodic output sequences with period greater than 2. Recall that an example of a FSA which has this period was previously mentioned. This indicates that the RCC network has only limited finite-state memory in the sense that it cannot represent certain cycles in a finite state automata. Again, an example of a FSA that RCC could not represent is Tomita-6, a FSA with a

3-cycle subcomponent.

It is arguable that the RCC network can have more complex dynamics during the transient period [31]. (For most simulations, this period is very short, usually a few time steps.) The transient period is the time before the neuron's activation value reaches its period point. But for the neural network to correctly represent the FSA, it must recognize arbitrarily long input strings. For an arbitrarily long input string, the transient period will have passed. The above proof does not depend on dynamical behavior during the transient period. (Of course, this ignores whether arbitrarily long strings can be correctly recognized. In our experience, this has been the case. Others have taken different measures to ensure this [43].)

It is important to notice that the monotone activation function is essential to the above proof. If the activity function is Gaussian or another non-monotonic function, then a more complicated behavior might be found. But since neurons early in the RCC growth do not have information feedback from later ones, the network's memory capacity will still be limited in following sense; the previously trained neurons will not have complete access to the state information learned by the later added neurons.

4.3 Numerical Simulations with RCC

Numerical simulations have shown the RCC network is unable to learn the simple parity grammar. The 2-state FSA that classifies strings of the parity language is shown in figure 7. This FSA recognizes all strings of arbitrary length that have an odd number of 1's; the objective is to train a recurrent network to do the same. When RCC is trained on even and odd strings of the parity grammar, the number of added neurons grows linearly (figure 7) with the longest string length of the training examples! Note that the proofs above say nothing about either of the FSA shown in figure 7 since both have period 1. Also note that the automaton labeled $fsa1$ does seem to be learnable by RCC!

5 Simple Expanding Recurrent Neural Network

The restriction that arose in the RCC is that the RCC constructive method does not preserve the necessary recursive connections. In order to get around this restriction, any constructive network must contain recursive connections to the earliest neurons. We propose a very simple constructive scheme to correct this problem and dynamically construct a recurrent network. Other more sophisticated methods could certainly be developed. In this method, the network is the same as for simple dynamically driven recurrent structure discussed in section 3. However, the recurrent layer is permitted to expand whenever needed and to keep its "full" connectivity. The only criterion for expansion is that the network spend some time learning. Previous work [24, 28] has shown that a large enough fully-connected, hard-threshold neuron recurrent network is capable of representing any finite state automaton and, more recently, with sigmoid-like neurons even more powerful

automaton [35]. As the size of network grows, so does the size of the FSA it can represent. Thus, any simple expanding network that is "fully-connected" and large enough should be able to represent (or load) any FSA [1, 15].

The importance of using prior knowledge in neural network learning has been described by many, see the discussion by Shavlik [34]. Our further assumption is that prior knowledge gained from early training is important to effectively maintain some of the network's earlier knowledge. To do this we require the network to be "expanded smoothly," *i.e.* the newly added weights should be zero or very small random numbers. The old weights start with their previously trained values, but are still trainable. Thus, the new network behaves very similarly to the old one immediately after the expansion and old knowledge learned is still preserved.

We present a simple example of the above method. We train a second-order fully-recurrent neural network [11] to be a FSA by training it sequentially on positive and negative strings accepted by that FSA. For training we use a true-gradient real-time recurrent learning algorithm (RTRL), for further description see [11]. First we examine if the network can take advantage of its previous knowledge. For Criteria 1) we require that the network train for some period of time. Admittedly, the amount of time could be crucial. However, this is an issue that remains to be examined. More complex decision criteria might be based on entropy measures, network capacity, neuron activity distribution or others.

We train the network to learn a 10-state randomly generated FSA (figure 8) as in [12]. The incremental real-time on-line training method permits the network to read through the ordered training samples until the accumulated error exceeds the preset value or all training samples are classified correctly. (Note that all FSA discussed here accept an arbitrarily number of strings.) When training with a fixed size 8 neuron fully-recurrent network with the initial weights randomly chosen, the network converged in 72 epoches after using 10000 positive and negative training examples. However, if first we train a similar network with only 7 recurrent neurons for 150 epoches and then expand the network to 8 recurrent neurons, the network takes only 32 more epoches to converge. Here an epoch is defined as a training cycle in which the network sees all training samples. (Keep in mind that there are many issues for training sequences, such as size and order of training set, and we do not attempt to address all of them.) This indicates that the network does take advantage of previous knowledge and converges faster than a network with no prior knowledge and random initial weights.

Choosing the criterion for when the network expands is very important. A simple method is to determine if the training error reaches a local minimum. However, this method has no guarantee of convergence and could lead to an explosive growth of neurons. This could be controlled by placing an upper bound on the number of grown neurons. For these simulations we only train for some short but arbitrary period of time.

In other preliminary experiments, we initially start with 2 fully-connected neurons and add one more neuron to the recurrent layer after every 50 epoches, until the network learns all the training samples. Thus,

if the resultant network has 5 neurons, it trained for 200 epochs. Again, this will probably generate a slightly larger network than that generated by a more sophisticated growth criterion. In training very small grammars (those of [39]), we found that the constructive method quickly converges when the network size grows up to the minimal size required. The convergence time was not significantly less than that for training corresponding fixed-size networks. This is understandable since for a very simple problem, the fixed size network can learn the training samples very quickly (assuming a large enough network is provided). However, by avoiding training all different size networks, the constructive method does appear to save time in finding the smallest size network for each of these grammars. Table 1 compares the minimum network size found by trail and error [36] for each of Tomita's grammars and the various sizes of the networks trained by our constructive method for 5 different runs with random initial conditions. The number of neurons required for the "trail and error" method and the constructive method described above is comparable. We would speculate that for the constructive method to get the minimal number of neurons obtained by "trail and error" method, a more sophisticated criteria for neuron growth should be used. It was surprising that the constructive method does as well as it does.

6 Conclusions

We proved that for both monotone (eg. sigmoid) and hard-threshold neurons the recurrent casade correlation (RCC) constructive method has a representational limitation. We experimentally illustrated this by attempting to learn the parity grammar. This restriction on RCC is topological and has no relation to its training algorithms.

We then presented some preliminary results on a simple constructive learning method that maintains the fully-connected recurrence necessary to eliminate the problems of RCC. (This fully-connected recurrence could be used in other more-sophisticated constructive methods.) Our method relies on using some of the knowledge of a partially trained neural network, and more importantly, expands the network in a fully recurrent manner.The criterion for expansion of the network is a simple one; training must occur for some short period of time. More sophisticated criteria could be used [8].

The recurrent cascade-correlation construction method is certainly a step in the right direction, but it is incapable of representing and thus learning many finite state automata. The simple constructive method proposed avoids the limitations of Recurrent Cascade Correlation (RCC) networks. We illustrate new this method by learning some small grammars of Tomita and a randomly generated 10-state grammar. The reason the simple constructive method outperforms RCC is that the full recurrence of the growing network is preserved. Admittedly, the experiments described are preliminary. Obviously, further experiments are necessary to better understand the criteria for expansion and the amount of knowledge necessary for "good"

learning.

Obviously, our constructive method requires further work. It seems not a good idea to simply add more neurons after a fixed number of epoches. To avoid having the network grow too large, neurons need to be added only when the network has shown difficulty in learning previously unseen new samples. It would most likely be useful to determine this dynamically, say during training. One obvious approach is to monitor the network's learning (error) curve. However, when using on-line training (vs. batch training), the error for different strings will dramatically change. It is also very difficult to find the saddle point of the curve. One possibility is to average over a long period to get a smoother curve. A weight connection with a very large magnitude is hard to correct, since the derivative of the sigmoid becomes very small at this point; and we do not want the small network to be over trained. So the averaging proposed above might make a few weight connections very strong and thus difficult to later correct.

7 Acknowledgements

The authors would like to acknowledge useful suggestions from T. Petsche and the reviewers. The University of Maryland authors gratefully acknowledge partial support from AFOSR and ARPA.

References

[1] N. Alon, A. Dewdney, and T. Ott, "Efficient simulation of finite automata by neural nets," *Journal of the Association for Computing Machinery*, vol. 38, no. 2, pp. 495–514, April 1991.

[2] T. Ash, "Dynamic node creation in backpropagation networks," *Connection Science*, vol. 1, no. 4, pp. 365–375, 1989.

[3] 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.

[4] J. Diederich, "Connectionist recruitment learning," in *Proceedings of the 8th European Conference on Artificial Intelligence*, (London, UK), 1988.

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

[6] S. Fahlman, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems 2* (D. Touretzky, ed.), (San Mateo, CA), pp. 524–532, Morgan Kaufmann Publishers, 1990.

[7] S. Fahlman, "The recurrent cascade-correlation architecture," in *Advances in Neural Information Processing Systems 3* (R. Lippmann, J. Moody, and D. Touretzky, eds.), (San Mateo, CA), pp. 190–196, Morgan Kaufmann Publishers, 1991.

[8] S. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," Tech. Rep. CMU-CS-90-100, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, February 1990.

[9] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Computation*, vol. 2, p. 198, 1990.

[10] S. I. Gallant, "Three constructive algorithms for network learning," in *Proceedings, 8th Annual Conference of the Cognitive Science Society*, pp. 652–660, 1986.

[11] 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, pp. 393–405, 1992.

[12] C. Giles, C. Miller, D. Chen, G. Sun, H. Chen, and Y. Lee, "Extracting and learning an unknown grammar with recurrent neural networks," in *Advances in Neural Information Processing Systems 4* (J. Moody, S. Hanson, and R. Lippmann, eds.), (San Mateo, CA), pp. 317–324, Morgan Kaufmann Publishers, 1992.

[13] C. Giles and C. Omlin, "Extraction, insertion and refinement of symbolic rules in dynamically-driven recurrent neural networks," *Connection Science*, vol. 5, no. 3,4, pp. 307–337, 1993. Special Issue on Architectures for Integrating Symbolic and Neural Processes.

[14] 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.), (Piscataway, NJ), pp. 13–22, IEEE Press, 1992.

[15] M. Goudreau, C. Giles, S. Chakradhar, and D. Chen, "First-order vs. second-order single layer recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 3, pp. 511–513, 1994.

[16] S. J. Hanson, "Meiosis networks," in *Advances in Neural Information Processing Systems*, vol. 2, pp. 533–541, 1990.

[17] S. A. Harp, T. Samad, and A. Guha, "Designing application-specific neural works using the genetic algorithm," in *Advances in Neural Information Processing Systems 2* (D. Touretzky, ed.), (San Mateo, CA), pp. 447–454, Morgan Kaufmann Publishers, 1990.

[18] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems 5* (S. Hanson, J. Cowan, and C. Giles, eds.), (San Mateo, CA), Morgan Kaufmann Publishers, 1993.

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

[20] Y. Hirose, K. Yamashita, and S. Hijiya, "Back-propagation algorithm which varies the number of hidden units," *Neural Networks*, vol. 4, pp. 61–66, 1991.

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

[22] D. Hush and B. Horne, "Progress in supervised neural networks," *IEEE Signal Processing Magazine*, vol. 10, no. 1, pp. 8–39, 1993.

[23] E. Jackson, *Perspectives of Nonlinear Dynamics*, vol. 1, p. 149. Cambridge, UK: Cambridge University Press, 1991.

[24] 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.

[25] M. Marchand, M. Golea, and P. Rujan, "A convergence theorem for sequential learning in two-layer perceptrons," *Europhysics Letters*, vol. 11, p. 487, 1990.

[26] M. Mézard and J.-P. Nadal, "Learning in feedforward layered networks: The tiling algorithm," *Journal of Physics*, vol. 21, p. 2191, 1989.

[27] C. Miller and C. Giles, "Experimental comparison of the effect of order in recurrent neural networks," *International Journal of Pattern Recognition and Artifical Intelligence*, vol. 7, no. 4, pp. 849–872, 1993. Special Issue on Neural Networks and Pattern Recognition, editors: I. Guyon , P.S.P. Wang.

[28] M. Minsky, *Computation: Finite and Infinite Machines.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1967. Ch: Neural Networks. Automata Made up of Parts.

[29] M. C. Mozer and P. Smolensky, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," *Connection Science*, vol. 11, pp. 3–26, 1989.

[30] O. Nerrand, P. Roussel-Ragot, G. D. L. Personnaz, and S. Marcos, "Neural networks and non-linear adaptive filtering: Unifying concepts and new algorithms," *Neural Computation*, vol. 5, pp. 165–197, 1993.

[31] T. Parker and L. Chua, *Practical Numerical Algorithms for Chaotic Systems*, p. 4. New York, N.Y.: Springer-Verlag, 1989.

[32] J. Pollack, "The induction of dynamical recognizers," *Machine Learning*, vol. 7, no. 2/3, pp. 227–252, 1991.

[33] D. L. Reilly, C. Scofield, C. Elbaum, and L. N. Cooper, "Learning system architetures composed of multiple learning modules," in *Proceedings of the IEEE First International Conference On Neural Networks*, (San Diego), 1987.

[34] J. W. Shavlik, "Combining symbolic and neural learning," *Machine Learning*, vol. 4, no. 3, p. 321, 1994.

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

[36] H. Siegelmann, E. Sontag, and C. Giles, "The complexity of language recognition by neural networks," in *Algorithms, Software, Architecture - Information Processing 92, Vol 1* (J. van Leeuwen, ed.), pp. 329–335, Amsterdam, The Netherlands: Elsevier Science, 1992.

[37] J.-A. Sirat and J.-P. Nadal, "Neural tree: A new tool for classification," tech. rep., Laboratories d'Electronique Philips, Limeil-Brevannes, France, 1990.

[38] S. Solla, "Capacity control in classifiers for pattern recognition," 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. 255–266, IEEE Press, 1992.

[39] 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.

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

[41] R. Williams and D. Zipser, "Experimental analysis of the real-time recurrent learning algorithm," *Connection Science*, vol. 1, no. 1, pp. 87–111, 1989.

[42] 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.

[43] Z. Zeng, R. Goodman, and P. Smyth, "Learning finite state machines with self-clustering recurrent networks," *Neural Computation*, vol. 5, no. 6, pp. 976–990, 1993.

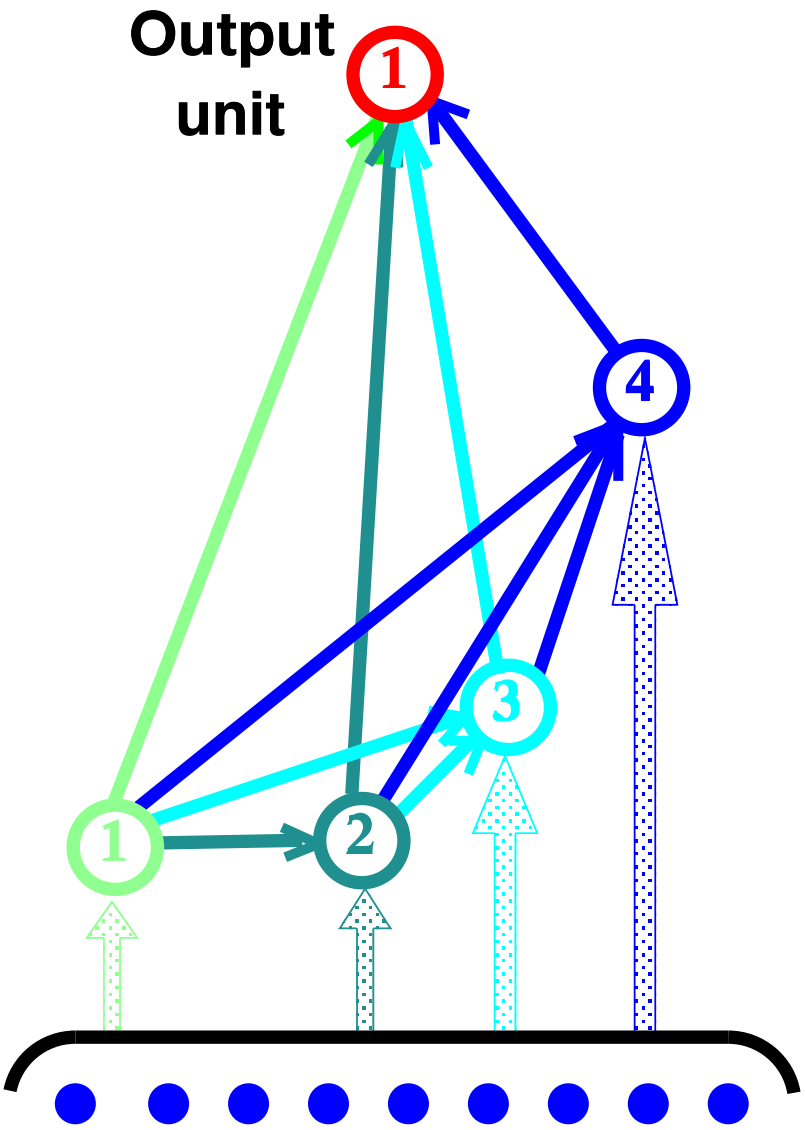

Figure 1: Cascade-Correlation (CC) Architecture. This feedforward architecture starts with one hidden unit and one output unit, both labelled 1. (In Fahlman's original CC architecture, there were initially no hidden units. We start with one so that the comparison of CC with RCC and the fully-recurrent constructive method is more instructive.) Arrows represent trainable weights. Hidden units are added one at a time in numerical order. For this example 3 hidden neurons are grown. The same previous inputs are also connected to the newly added hidden units as they are formed.

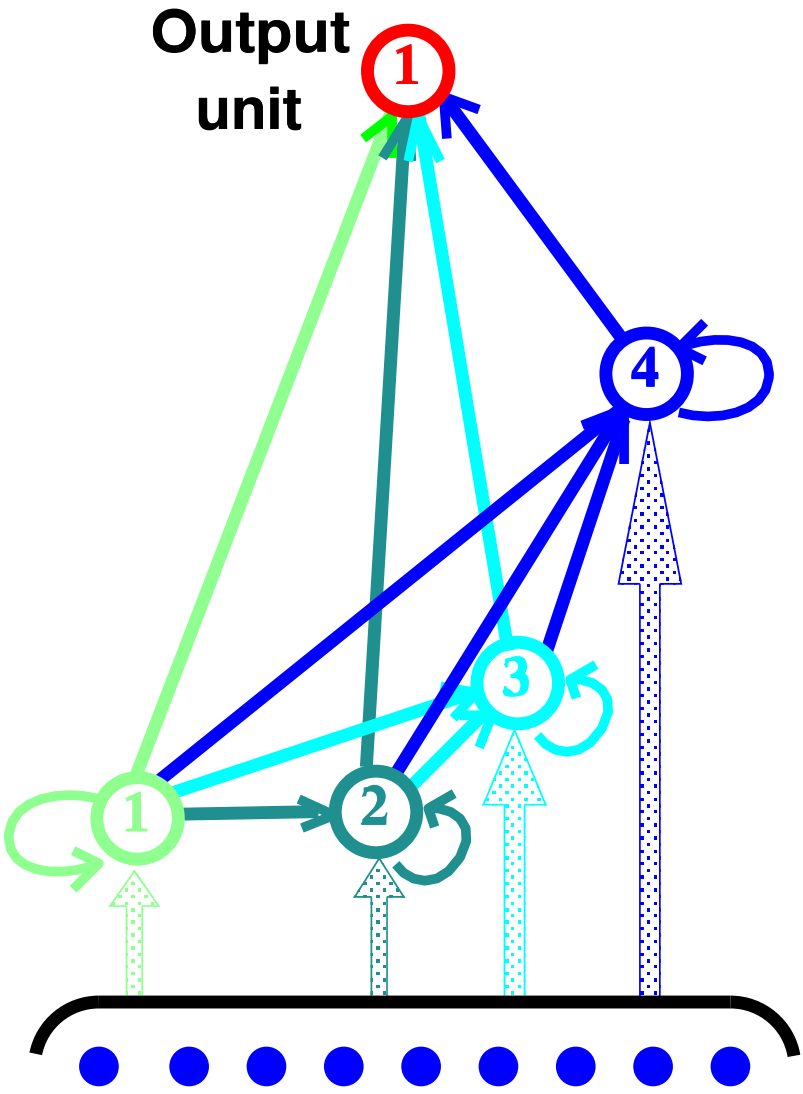

Figure 2: Recurrent Cascade-Correlation Network. As in the Cascade-Correlation Figure, the network starts with neuron 1 and its appropriate output unit. In this example RCC grows 3 neurons one step at at time. The hidden neurons are self-recurrent and only connect in one direction to previous existing neurons. Arrows represent trainable weights.

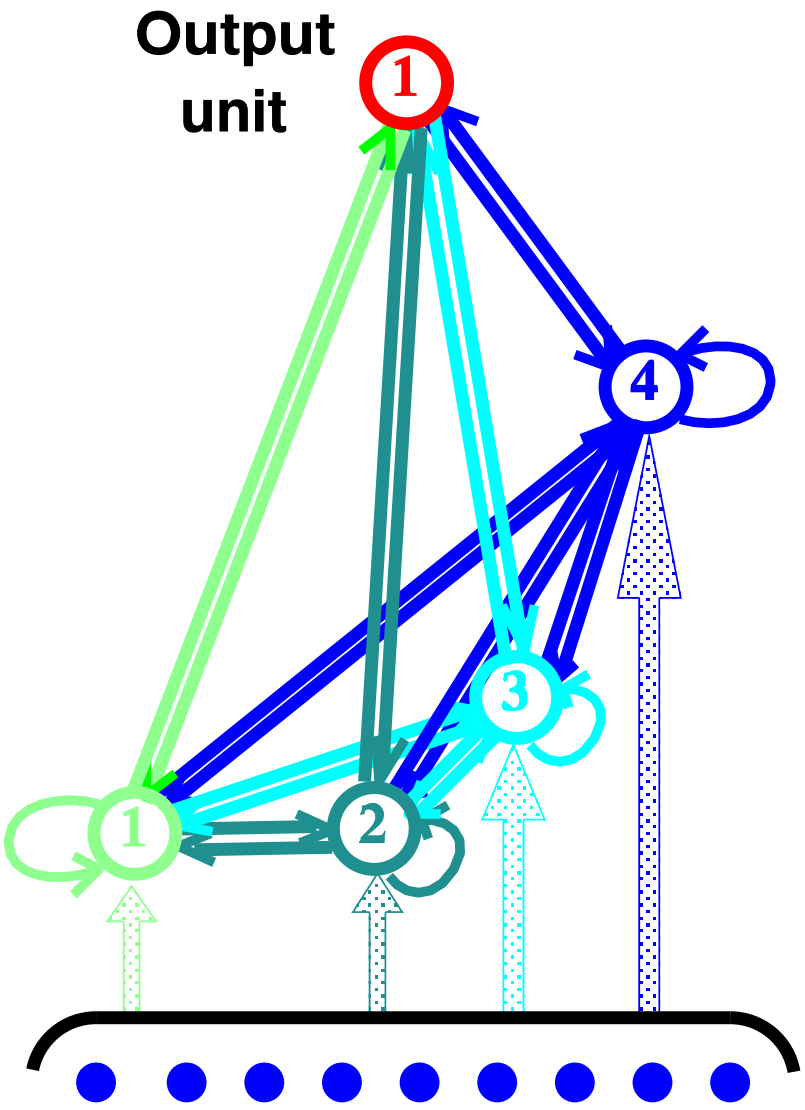

Figure 3: Expanding Fully-recurrent Neural Network. The input and output neurons are connected in both directions to all hidden recurrent neurons. Again, as in the Cascade-Correlation Figure, the network starts with neuron 1 and grows incrementally to 4 neurons. Arrows represent trainable weights.

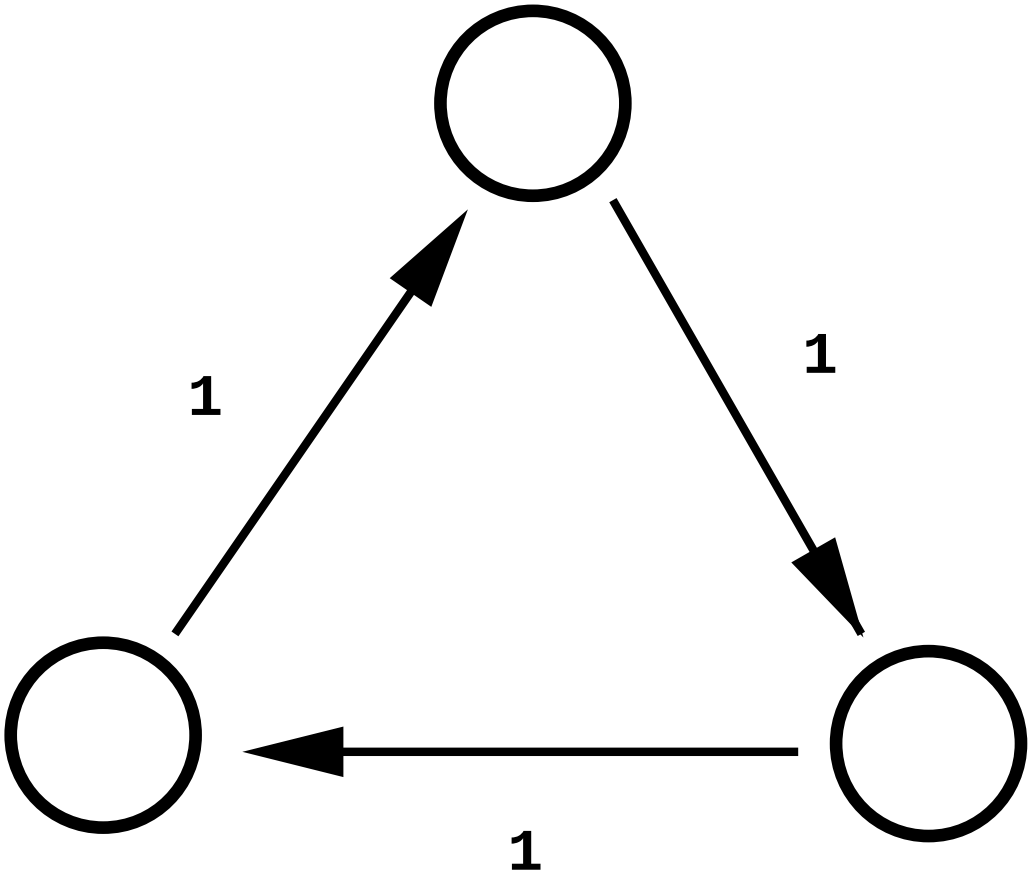

Figure 4: 3-cycle Finite State Automaton (FSA) Subcomponent. An example of a 3-cycle subcomponent of a FSA that RCC cannot represent. There are 3 states each connected by the input alphabet character 1.

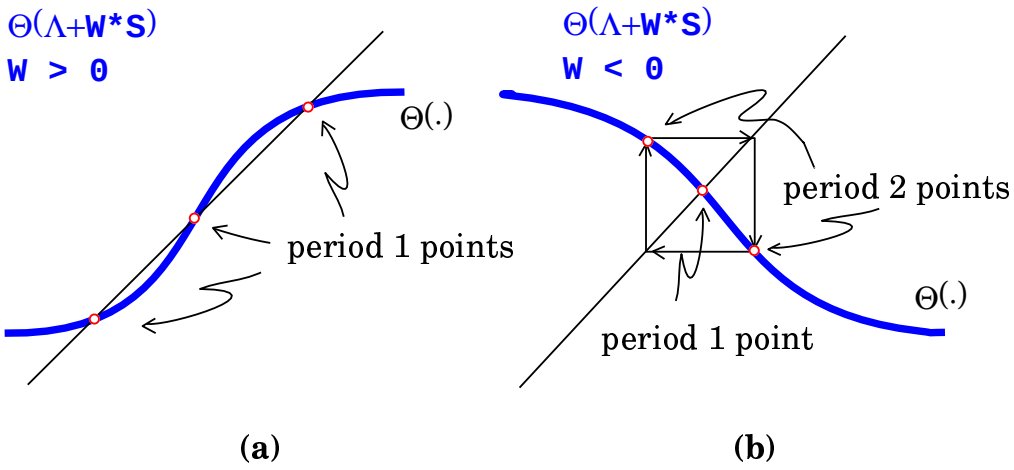

Figure 5: Period Points of a Sigmoid Activation Function. When $W > 0$, the function $\Theta(.)$ monotonically increases and has at most 3 period-1 points. When $W < 0$, $\Theta(.)$ monotonically decreases and has two period-2 points and one period-1 point.

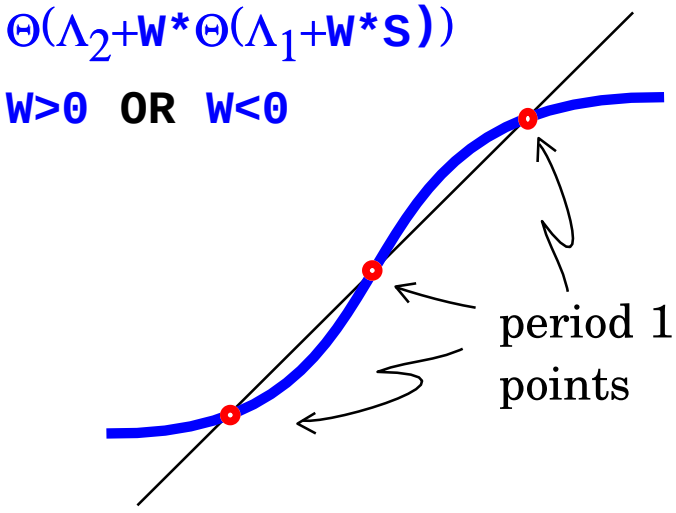

Figure 6: Period Points of the Recurrent Activations $\Theta(\Theta(.))$. Regardless of $W's$ value, the function $\Theta(\Theta(.))$tr monotonically increases and has at most 3 period-1 points.

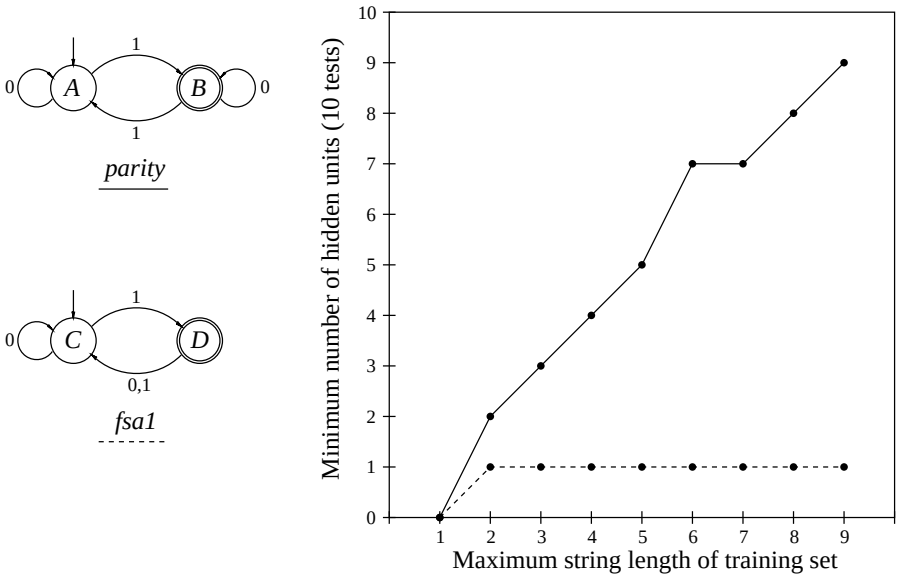

Figure 7: RCC Learning of 2-state Finite State Automata (FSA). The minimum number of hidden units is plotted versus of the maximum string length of all strings used in training. The FSA are the parity automaton which accepts all strings with an odd number of 1's and the automaton that accepts all strings that end in a single 1.

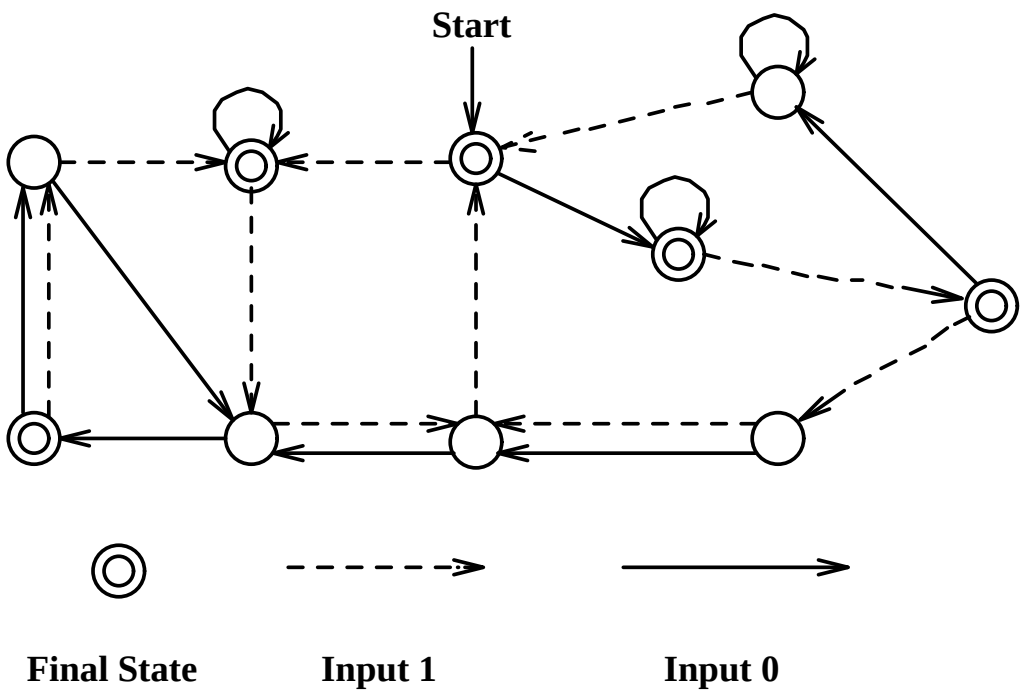

Figure 8: A Randomly Generated 10-state Finite State Automaton. The dashed and solid arrows are for inputs of 1 and 0 respectively. The double circles are accepting states.

FSA	minimum neuron size found by "trial and error"	neuron sizes found by contructive method
$1*$	1	3, 3, 3, 3, 2
$(10)*$	2	3, 3, 3, 2, 3
no (1^{odd}) followed by (0^{odd})	3,	4, 3, 4, 4, 4
no (000)	2	4, 5, 5, 5, 3,
$([01+10][01+10])*$	4	8, 5, 6, 7, 8
$\sharp 1 - \sharp 0 = 3k$	3	7, 4, 4, 3, 7
$0*1*0*1*$	2	6, 5, 5, 6, 4

Table 1: FSA versus Number of Neurons. The left most column is the FSA to be learned. The center column is the minimum number of neurons needed to learn this FSA using a "trail and error" method (see reference in text). The right column is the number of neurons needed to learn the FSA for 5 different randomly initialized runs for the simple contructive method discussed in the text.