Learning Large DeBruijn Automata with Feed-Forward Neural Networks

University of California, San Diego
Computer Science and Engineering Technical Report CS94-398

Daniel S. Clouse* C. Lee Giles† Bill G. Horne‡ Garrison W. Cottrell§

December 22, 1994

Abstract

In this paper we argue that a class of finite state machines (FSMs) which is representable by the NNFIR (Neural Network Finite Impulse Response) architecture is equivalent to the definite memory sequential machines (DMMs) which are implementations of deBruijn automata. We support this claim by drawing parallels between circuit topologies of sequential machines used to implement FSMs and the architecture of the NNFIR. Further support is provided by simulation results that show that a NNFIR architecture is able to learn perfectly a large definite memory machine (2048 states) with very few training examples. We also discuss the effects that variations in the NNFIR architecture have on the class of problems easily learnable by the network.

*UCSD CSE Dept., dclouse@ucsd.edu
†NEC Research Institute, Princeton NJ, giles@research.nj.nec.com
‡NEC Research Institute, Princeton NJ, horne@research.nj.nec.com
§UCSD CSE Dept., gcottrell@ucsd.edu

1 Introduction

In the neural network language induction literature, induction of finite state automata is commonly thought of as the domain of recurrent network architectures [Cleeremans et al., 1989] [Elman, 1991] [Pollack, 1991] [Giles et al., 1992] [Watrous and Kuhn, 1992]. However, recent work [Giles et al., 1994] has shown that a restricted class of recurrent nets can learn a subclass of finite state automata called finite-memory automata. In this paper, we show that *feedforward-only* architectures can represent and learn a class of automata, DeBruijn automata. These automata accept strings of arbitrary length, but have states that are completely determined by a finite length of input strings. As such these automata can be implemented by a class of sequential machines termed *Definite Memory Machines* (DMMs). Structurally these DMMs resemble time-delay neural networks (TDNNs). TDNNs were originally defined [Waibel et al., 1989], to include feedforward nets with both input time-delays and internal time-delays. Like TDNNs, DMMs make no distinction between these two forms of time delays either. In the literature [Wan, 1993], a TDNN is also referred to as an NNFIR (Neural Network Finite Impulse Response) architecture. For clarity we define the subclass which has only time delays on the inputs as an Input Delay Neural Network (IDNN).

In this paper we discuss the capability of both the NNFIR and IDNN architectures for the language induction task. We will find that even though the NNFIR and IDNN architectures are both capable of representing the same set of functions there is a discernable difference in their learning capabilites. In spite of the fact that these network architectures contain no recurrence, they can do quite well at induction of definite memory machines, a subclass of finite state machines.

2 Description of NNFIR Architecture

In a feed-forward neural network, the nodes can be ordered such that the inputs to any node i are the outputs of nodes 1, 2, ..., i-1. Using this formulation, the activation function for some node i in a multi-layer perceptron (a feed-forward network whose nodes are perceptrons) is given by equation 1.

$$y_i = h(\sum_{j=1}^{i-1} w_{ij} y_j) \tag{1}$$

where y_i is the output of node i, w_{ij} is the connection strength from node j to node i, and h is the squashing function. If there are p inputs to the network, then the first p nodes are used as input units

$$\forall i \leq p, y_i = u_i$$

where u_i is the i^{th} network input.

An NNFIR (Neural Network Finite Impulse Response [Wan, 1993]; also known as a Time Delay Neural Network or TDNN [Lang et al., 1990]) is similar to a multi-layer perceptron

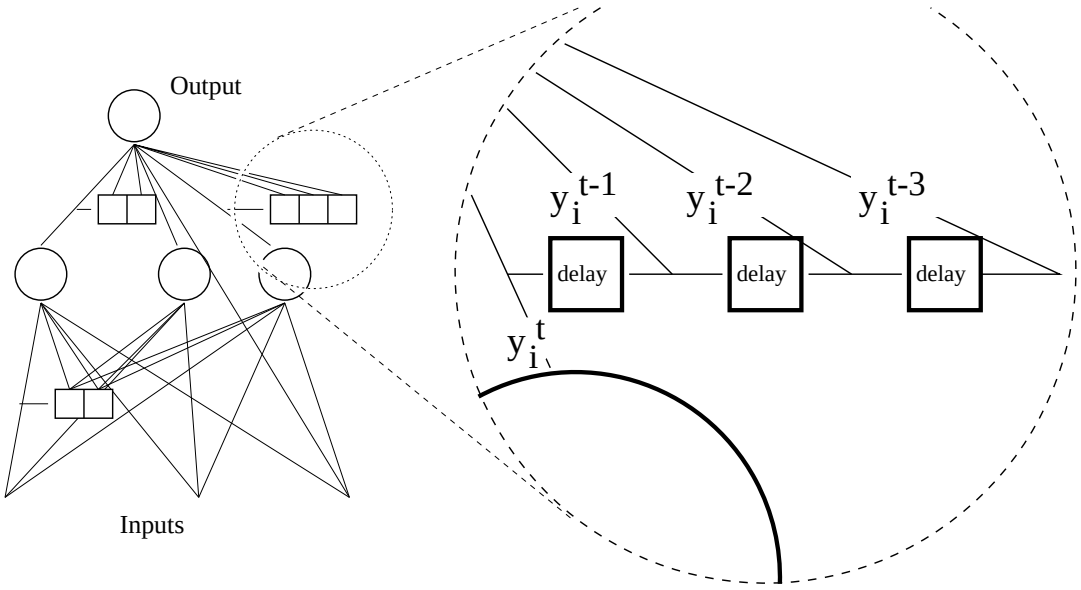

Figure 1: An NNFIR Network

in that all connections feed forward. The difference is that in the NNFIR, the inputs to a node i consist of the outputs of earlier nodes not only during the current time step t, but during some number, d, of previous time steps $(t-1, t-2, \ldots t-d)$ as well. The activation function for node i at time t in such a network is given by equation 2.

$$y_i^t = h(\sum_{j=1}^{i-1} \sum_{k=0}^{d} y_j^{t-k} w_{ijk}) \tag{2}$$

where w_{ijk} is the connection strength to node i from the output of node j at time k. Notice that the number of weights increases linearly with the number of delays. There are now $d+1$ weights for every pair of nodes in the network, one weight for each time delay step.

The NNFIR is generally implemented using tapped delay lines. For each node in the network, a tapped delay line stores the output of the node during the previous d time steps. The output of a node along with the values of the associated tapped delay lines at all delay intervals serve as input to later nodes. Figure 1 shows an example of an NNFIR network.

The length of the tapped delay lines determines the range of input history to which the network is sensitive. Since, in the NNFIR architecture, the delay size is fixed, the network will not be able to represent mappings which require a longer input history. To our knowledge, there has been no work on adapting the number of delays, but techniques for adapting the number of time steps between delays have been developed [Pearlmutter, 1989], [Lin et al., 1992], [Day and Davenport, 1993].

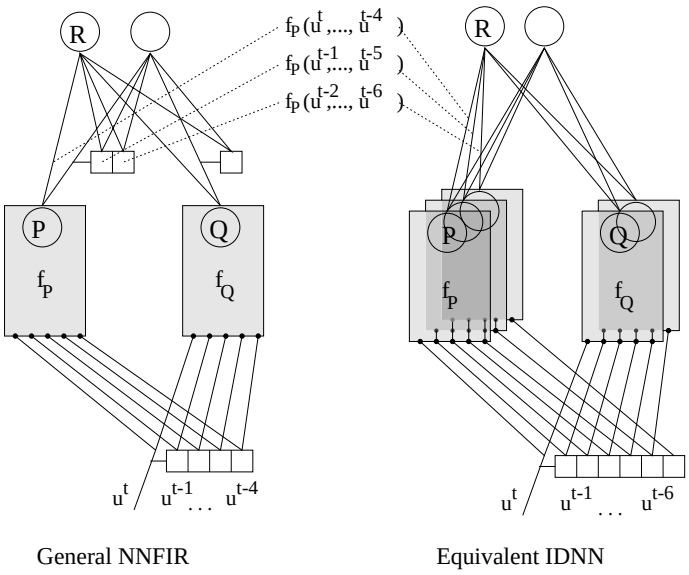

Figure 2: Transforming an NNFIR into an equivalent IDNN

3 Description of IDNN Architecture

A natural restriction of the general NNFIR topology is the class of NNFIR architectures which have delays only on the input units. We call these *input delayed neural networks* (IDNNs). An IDNN can be implemented with a single tapped delay line for each input. Each tapped delay line contains its inputs from the previous d time steps. These delayed inputs feed into a normal feed forward network. Because the tapped delay lines are effectively separate from the feed forward network, training of IDNNs is equivalent to training a feed forward network.

Since the class of IDNNs is a subclass of NNFIRs, every IDNN is also an NNFIR. Clearly then, the set of functions computable by the NNFIRs must include all those computable by the IDNNs.

It is less intuitive but also easy to show that the set of functions computable by any IDNN includes all those computable by the general NNFIR class [Wan, 1993][1]. Figure 2 and the discussion below presents a construction which results in removing all the internal delay lines from an NNFIR network and replacing them with an increased number of delays on the network inputs, resulting in an IDNN network which computes the same output function.

Consider any node R in some NNFIR. R computes a function of its direct inputs. Some of these inputs come from tapped delay lines and some are the immediate outputs of earlier nodes in the network. The output of each earlier node in the network can be written as a function of the network inputs at various amounts of delay. (For simplicity of notation,

[1]We know of no proof which covers the case of short input strings which do not fill the delay buffers. In this case, it is not clear that the IDNN and NNFIR architectures are equivalent.

in this discussion we will assume a single network input. The argument is easily extensible to larger numbers of inputs.) If d is the number of delays on the network input then the function computed by some earlier network node, P, can be written as $f_P(u^t, \ldots, u^{t-d})$. The output of the k^{th} delay of this node is then $f_P(u^{t-k}, \ldots, u^{t-d-k})$. In the original version of the network $f_P(u^{t-k}, \ldots, u^{t-d-k})$ is computed by delaying the output of node P, but the notation used here makes it clear that this function can also be computed by applying function f_P to a delayed set of inputs. In the network, this corresponds to making an exact copy of P and all the nodes, delays and weights feeding into P and sliding this subnetwork down the input delay lines so that all of its inputs are delayed by k time steps. This construction, in effect, replaces the delays on the output of node P by adding more delays on the network input.

We can make a similar transformation to all of the nodes which feed into node R, thus removing all reliance of node R on any direct input delays except for possible direct input delays from the network inputs. This same construction can be applied to any node in the resulting transformed network which will eventually result in removal of all internal delays from the network. The NNFIR network is thus changed into a functionally equivalent IDNN network.

In the next section, we show that the IDNN network is capable of representing a subclass of the FSMs. The functional equivalence of IDNNs and NNFIRs will allow us to extend this result to the NNFIRs.

4 Relationship Between Sequential Machines, DMMs, and NNFIRs

In order to understand the representational capabilities of the NNFIR architecture, we develop an argument based on the concept of sequential machines from VLSI design. A sequential machine is an implementation of a finite state machine (FSM) in digital logic and delays [Kohavi, 1978]. The sequential machine formalism separates combinational logic from memory elements. In a sequential machine implementation of an FSM, the memory elements contain the current state of the FSM. Combinational logic is used to define the state transitions and the outputs of the machine.

An FSM is said to be of input-order d if d is the least integer such that the present state of the FSM can always be determined uniquely from the knowledge of the last d inputs. Such an FSM, whose state depends only on a fixed number of recent inputs, is called a *definite memory machine* (DMM)[Kohavi, 1978, Chapter 10].

Since the current state of a DMM can be derived from the most recent d inputs, any DMM can be implemented as a sequential machine whose only memory elements are tapped delay lines on the input.

It is a short step from a sequential machine implementation of a DMM to an IDNN implementation of a DMM. Just replace the combinational logic in the sequential machine with a feed forward neural network. If the inputs and outputs of an IDNN are constrained

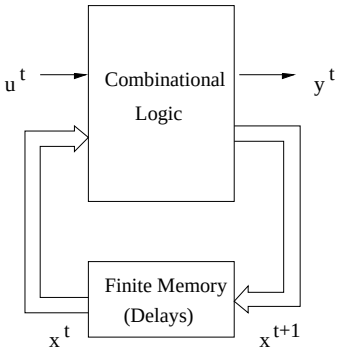

Figure 3: A Sequential Machine

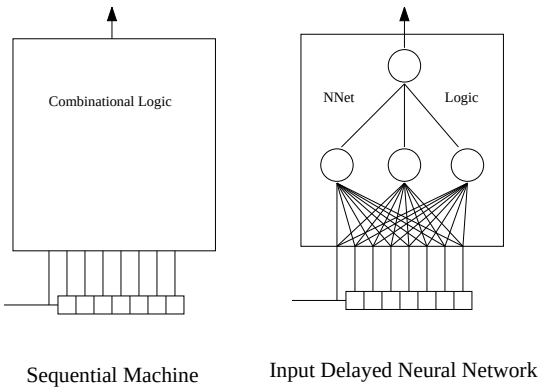

Figure 4: Any DMM can be implemented as an IDNN

to 1s and 0s IDNNs implement exactly the DMMs. We have already seen that the NNFIR and IDNN classes are functionally equivalent. This implies that NNFIRs implement DMMs as well.

On the other hand, if the inputs and outputs are allowed to take on real values, problems not representable by a finite state automata can be effectively handled [Wan, 1993]. Still, the finite history maintained by the NNFIR architecture precludes the representation of some regular languages. For example, the language 10* requires that the information that a 1 has been seen to be maintained across a potentially infinite number of 0s. This kind of infinite history cannot be maintained by the tapped delay lines of the NNFIR.

The IDNN architecture is a natural fit for the problem of inducing DMMs from labeled input strings. In the following section, we support this claim by learning a 2048 state DMM with an IDNN. Later, we discuss the bias of other variations on the general NNFIR architecture.

5 Learning a large DMM with little logic

Because of the close relationship between the IDNN architecture and DMMs, it is possible to learn DMMs with many states using the IDNN architecture. In this section we present the results of just such an experiment.

The machine learned is a DMM of input order 11. In its sequential machine form, it corresponds to the logic function of equation 3. Here the symbol $\leftrightarrow$ represents the if-and-only-if function.

$$y_k = u_{k-10} \leftrightarrow \overline{u}_k \overline{u}_{k-1} \overline{u}_{k-2} + \overline{u}_{k-2} u_{k-3} + u_{k-1} u_{k-2} \tag{3}$$

The logic function above was chosen so that its minimal FSM representation would require $2^{11} = 2048$ states. We discuss the conditions necessary for minimality in the next section, and in the appendix. The transition diagram for the minimal FSM is shown in figure 5.

To create training and test sets, we generated all strings of length 1 to 11, 4094 in all, and labeled them with a 1 or 0 depending on whether the DMM would accept or reject them. A trial consisted of a randomly chosen percentage of these strings on which the network was trained to the error criterion described below.

The network architecture consists of an IDNN with 10 tapped delays on the input. The multi-layer perceptron part of the architecture contains seven hidden nodes and a single output node. Each node uses the standard asymmetric sigmoid nonlinearity. Full connections are supplied between all delays (including the undelayed input) and all the nodes of the hidden layer. Likewise, the network contains full connections between the hidden layer and the output.

During training (and testing), before introduction of a new string, the values of all the delays were set to 0. Online back propagation [Rumelhart et al., 1986] with a learning rate of 0.25 and momentum of 0.25 was used for training. Weight decay [Krogh and Hertz, 1992] with a weight decay parameter of 0.0001 was used.

A selective updating scheme was applied whereby weights were updated in an online fashion, but only if the absolute error on the current training sample was greater than 0.2. This effectively speeds up the algorithm by avoiding gradient calculations for any string which will result in very small weight adjustments.

Training was stopped when all examples in the training set yielded an absolute error of less than 0.2. In this experiment, this level of accuracy was generally achieved in 200 epochs or fewer. No trial required more than 4000 epochs of training.

Figure 6 plots the average classification error on the testing set for various sizes of training sets. We call this generalization error because it reports error only on examples not seen during training. For purposes of this graph the classification of the network was considered a 1 if the network output was greater than 0.5, and a 0 if output was less than

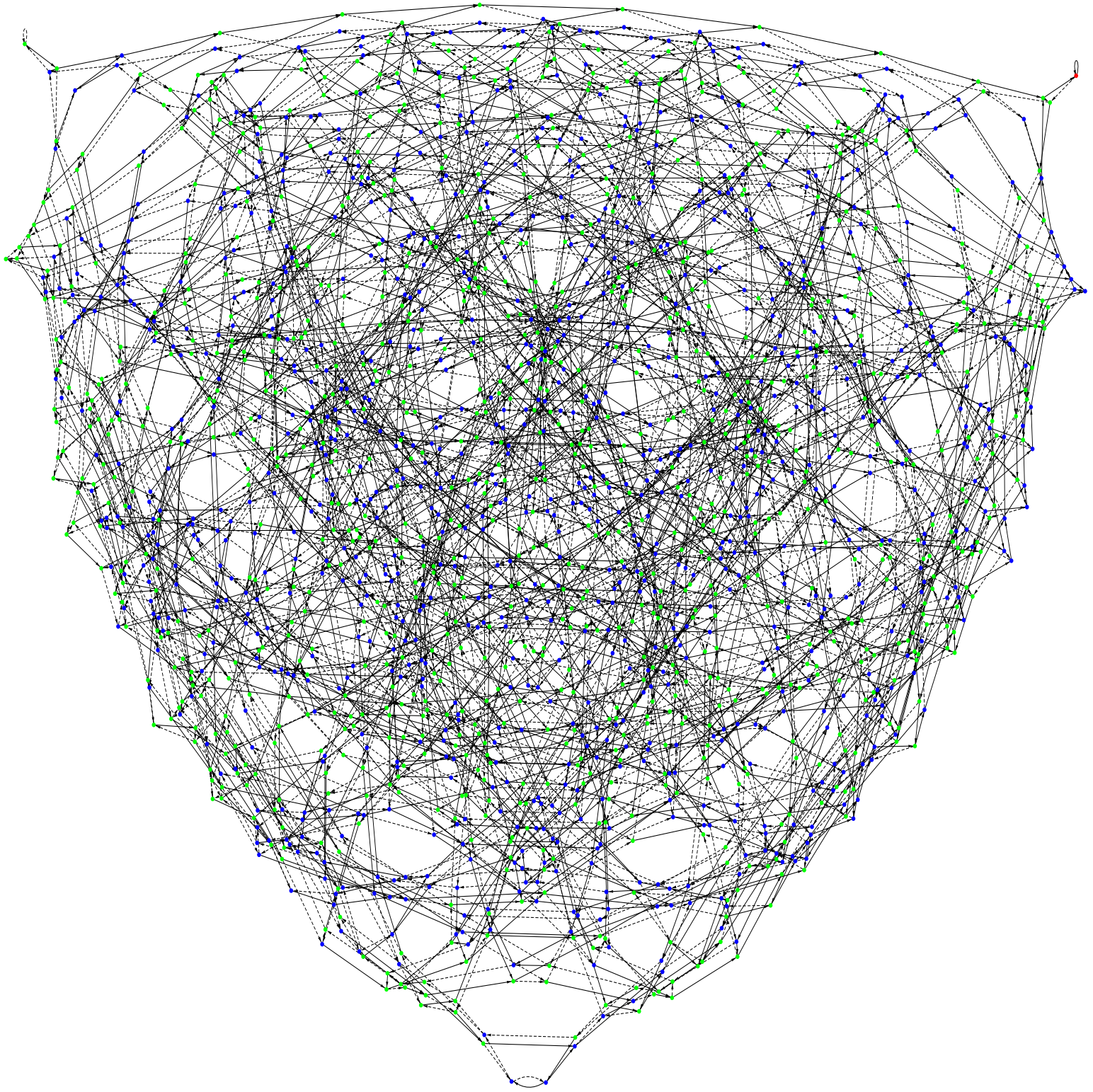

Figure 5: Transition diagram for 2048 state FSM

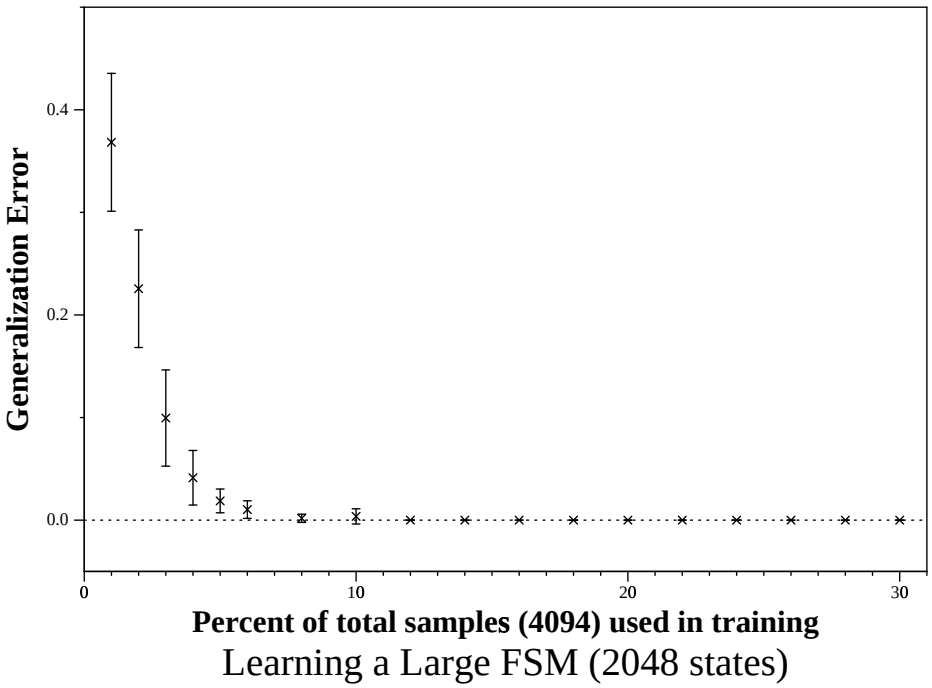

Figure 6: Generalization as a function of training set size on the 2048 state DMM.

or equal to 0.5. Therefore, a value of 0.5 on the graph indicates chance performance. The values plotted here are averaged across all strings in the test set, and across 20 trials using different randomly chosen initial weight parameters and randomly chosen training sets. The error bars indicate one standard deviation on each side of the mean calculated across the 20 trials.

It is clear from this figure that the IDNN is able to learn this particular large DMM quite accurately using only a small percentage of the potential input strings. However, the logic of the sequential machine implementation of the DMM is actually quite small; not all of the inputs were even used! In this sense one could interpret these results as best-case.

6 Why DMMs are Easily Learnable by an NNFIR

It is not difficult to understand why the IDNN is able to learn a large DMM if we think of the IDNN as a sequential machine. First, let us look at the transition diagram of a DMM as implemented by a sequential machine. In the sequential machine implementation of our DMM, the state of the machine is entirely determined by the contents of the input tapped delay line and the current input. The transition diagram is essentially that of a shift register. Such a transition diagram is called a directed binary deBruijn graph [Golomb, 1982].

Definition of Directed Binary DeBruijn Graph The *directed binary deBruijn graph* (henceforth deBruijn graph) of diameter d is the graph with 2^d nodes connected in a specific

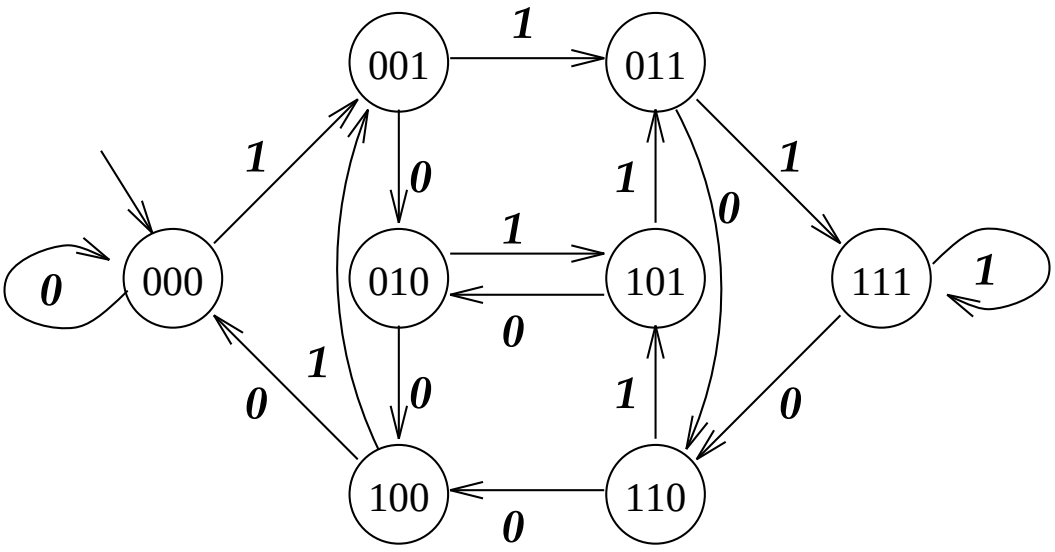

Figure 7: The DeBruijn Graph of Diameter 3

pattern. Each node is named with a unique string of d 0s and 1s. Node $< x_1x_2x_3 \ldots x_d >$ is connected to node $< x_1x_2 \ldots x_d0 >$ with an arc labeled with a 0, and to node $< x_1x_2 \ldots x_d1 >$ with an arc labeled with a 1.

The transition diagram of a deBruijn graph is completely determined by a single integer parameter, its diameter. The diameter of the deBruijn graph associated with a DMM implemented with n delays is $n + 1$. Figure 7 shows the deBruijn graph for a tapped delay line with 2 delays.

A deBruijn graph determines a transition diagram. By assigning accepting and rejecting states to the nodes of the graph, a number of DMMs can be generated. In the problem presented in the previous section, the function the network was required to learn was chosen so that it would map onto a deBruijn graph of diameter 11, and all $2^{11} = 2048$ states of the deBruijn graph would be necessary to represent the function. In other words, the 2048 state DMM is minimal.

The condition which guarantees the minimality of this DMM is that for any pair of nodes in the associated deBruijn graph whose names are identical except for the leftmost digit, one and only one node of this pair is an accepting state. This statement is presented as a theorem in the appendix, along with its proof.

Since there are 10 tapped delay lines and a single current input, the architecture of the IDNN is also predetermined to implement a deBruijn graph of diameter 11. The architecture and the problem to be learned have been chosen to have matching transition diagrams. Consequently, the IDNN architecture does not have to learn the state transitions at all.

Initializing all the delays to zeros before each pattern is presented, as we did in the simulation of the previous section, is equivalent to fixing the start state of the DMM to the

state whose name is all zeros. It is possible to allow the network to learn the start state by back-propagating error into the tapped delay lines, then initializing the delays with the altered values before a new pattern is presented. In this paper, all the problems attempted can be represented by a DMM with a zero start state.

Even though much of the problem representation is fixed by the IDNN architecture, a mapping from the current state to the correct accept/reject output must be learned. In essence, the problem presented to the multi-layer neural network portion of our architecture is that of learning the logic function which maps state number to output. For the DMM used in this experiment, the logic function is apparently simple enough that the network has little problem learning it perfectly.

Notice that it is simple to increase the number of states in the DMM by changing a single term of the mapping function. Instead of learning the function of equation 3 as we did in the previous section, we could train a network to learn equation 4.

$$y_k = u_{k-10} \leftrightarrow \overline{u}_k \overline{u}_{k-1} \overline{u}_{k-2} + \overline{u}_{k-2} u_{k-3} + u_{k-1} u_{k-2} \tag{3}$$

$$y_k = u_{k-11} \leftrightarrow \overline{u}_k \overline{u}_{k-1} \overline{u}_{k-2} + \overline{u}_{k-2} u_{k-3} + u_{k-1} u_{k-2} \tag{4}$$

This equation is identical except that it requires an extra input delay in the network. To represent this machine using a DMM will require a minimum of 4096 states, twice as many as the problem of the last section. Since the mapping function has essentially the same form as the problem of the earlier problem, we would expect the IDNN to perform about as well on this problem as it did on the earlier one. This demonstrates that the number of states in the DMM is not a good predictor of the difficulty of learning.

7 Learning a problem suited to a general NNFIR

Earlier we showed that the IDNN architecture and the general NNFIR architecture are functionally equivalent. That these two architectures are capable of *representing* the same class of functions does not imply that the two architectures are equally capable of *learning* the same set of functions. In this and the following sections, we present our intuitions about what kinds of functions each architecture may be good at learning. We provide support for these conjectures by running simulations on comparably-sized IDNN and NNFIR architectures.

Each of the architectures we use in the following simulations contains a single input and a single output. The IDNN architecture contains seven input delays, and two hidden layers each with four units. Like the IDNN architecture, the NNFIR architecture also contains two hidden layers, though in the NNFIR architecture, each hidden layer has only three nodes. The NNFIR architecture contains four input delays and three delays on the outputs of each node of the first hidden layer. Figure 8 illustrates the two architectures.

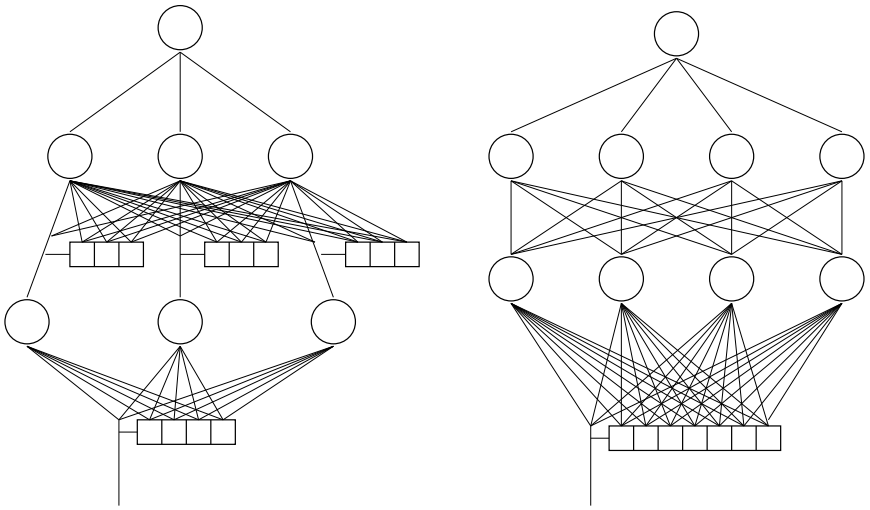

Figure 8: Architectures Used in Experiments

Note that in each architecture, the output node computes a function of the current input along with seven input delays. Also, each architecture contains four layers, and approximately the same number of weights (54 in the NNFIR, and 52 in the IDNN).

7.1 Learning repeated (systematic) logic terms

Our intuition suggests that the general NNFIR architecture may be better at learning DMM problems in which the function which maps state numbers to accepting and rejecting states contains terms which repeat across time. This intuition comes from viewing each node in the first hidden layer of the NNFIR architecture as developing a feature detector which looks for a specific pattern in its input window during recent time. During presentation of a string, a feature detector in the first hidden layer is able to detect this same repeated pattern in any time window. Later layers can combine the outputs of all these first layer feature detectors across a longer window of time because they have access to the outputs of the feature detectors at a number of delays.

To test this hypothesis, we ran a series of simulations which contrast the performance of an IDNN network with an NNFIR network in learning a DMM defined by a mapping function with repeated terms. The mapping function for the specific problem we chose to test is given in equation 5.

$$y_k = u_{k-7} \leftrightarrow u_k u_{k-2} \overline{u}_{k-3} + u_{k-1} u_{k-3} \overline{u}_{k-4} + u_{k-2} u_{k-4} \overline{u}_{k-5} + u_{k-3} u_{k-5} \overline{u}_{k-6} \qquad (5)$$

This function contains the same term four times, shifted in time. For this reason, we call this logic "systematic". The function has been chosen to meet the conditions of the

theorem presented in the appendix, which guarantees that 256 states are required in the minimal FSM representation. Through running many simulations, we have been able to find a set of weights which produces this functional behavior in each of the architectures, so we know that this function is representable by both the IDNN and NNFIR architectures.

The training method used for the IDNN network was identical to the simulation presented earlier except training was stopped after 7000 epochs even if perfect performance on the training set was not achieved. This early stopping is necessary to avoid infinite training when a suboptimal local minimum has been reached.

Training for the NNFIR architecture is slightly more complicated. Error from the final output must be propagated backwards across the internal tapped delay lines. This involves propagating the error from the final output back to each delay in the internal tapped delay line, then shifting the resulting buffer of errors one step backwards for each earlier input step. When an error value reaches the beginning of the buffer it serves as the error for the node which drives the associated delay line. See [Wan, 1993] for a detailed description of the general algorithm. With the exception of the difference of this more complicated training method, all details of the NNFIR training were identical to that of the IDNN training.

The results of the simulation are presented in figure 9. Plotted points are the mean classification error for the IDNN and NNFIR architectures averaged across 20 trials at each training set size. The mean performance of the NNFIR architecture across the entire curve is significantly better than that of the IDNN architecture as shown by a two-way analysis of variance (ANOVA) [Rice, 1988] ($MS_{arch} = 0.2749$, $MS_{error} = 0.0027$, $F(1, 456) = 101.427$, $p < 0.001$)[2]. This result supports our hypothesis that the general NNFIR architecture is better at learning a DMM whose mapping function contains repeated terms. From the graph, the separation is especially pronounced for large training set size. However, for very small training set sizes, it appears that the two architectures perform about the same.

7.2 Learning an equal state FMM with non-systematic logic terms

To provide further evidence, we ran a similar simulation contrasting the performance of the two architectures in learning a DMM whose mapping function does not contain contain repeated terms but has the same number of 256 states. We call this logic "non-systematic" because it lacks repeated terms. The mapping function for this problem is given in equation 6.

$$y_k = u_{k-7} \leftrightarrow u_k u_{k-2} \overline{u}_{k-3} + u_{k-1} \overline{u}_{k-4} \overline{u}_{k-6} + \overline{u}_{k-3} u_{k-4} \overline{u}_{k-6} \tag{6}$$

The architectures and training methods used in this simulation were identical to those of

[2] MS_{arch} and MS_{error} are two independent estimates of the variance of the data. One estimate, MS_{arch} relies on the null hypothesis, that there is no effect of the difference in architectures on the error rate. The other estimate, MS_{error}, is independent of this assumption. If the null hypothesis is true, then F, the ratio between these two estimates, is expected to be close to 1. As it stands, our F value is much greater than 1. The probability of getting an F value this large, when the null hypothesis is true, is given by p. Since this probability is small, we are justified in rejecting the null hypothesis.

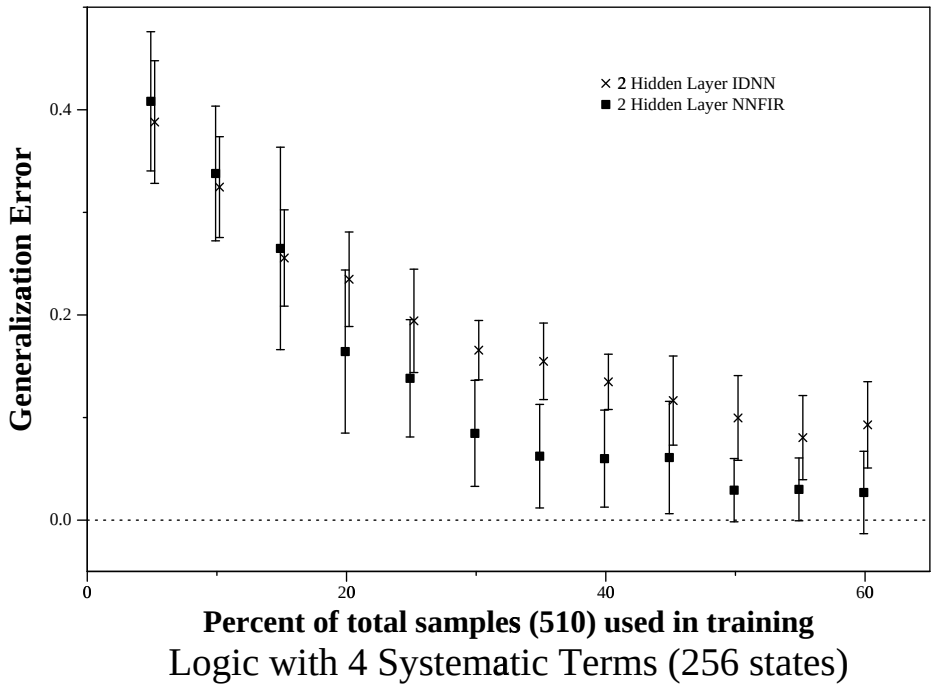

Figure 9: Generalization performance of a general NNFIR and an IDNN on a systematic problem

the previous simulation. The results are presented in figure 10. Here we see that even though the performance of the two networks is very similar, the IDNN network outperforms the NNFIR network at every test point. The difference is indeed significant ($MS_{arch} = 0.0717$, $MS_{error} = 0.0016$, $F(1, 456) = 45.753$, $p < 0.001$). This suggests that the IDNN network has a slight advantage over the NNFIR network on these kinds of problems without repeated terms.

7.3 Learning wide logic terms

Another hypothesis is that the NNFIR architecture does well only with mapping functions in which each term is capable of fitting in the input window of a single first layer feature detector. If this hypothesis is true, we would expect the NNFIR architecture to perform poorly on a problem in which the mapping function had wide repeated terms. We ran a final simulation on just such a problem. The mapping function for this problem is given in equation 7.

$$y_k = u_{k-6} \leftrightarrow u_k u_{k-2} \overline{u}_{k-3} + u_{k-1} u_{k-3} \overline{u}_{k-4} + u_{k-2} u_{k-4} \overline{u}_{k-5} \tag{7}$$

We deleted two input delays from the NNFIR architecture used in the earlier experiments along with the associated connections to the first hidden layer. This narrowing of the input window was done to make sure that no term of the mapping function would fit into the

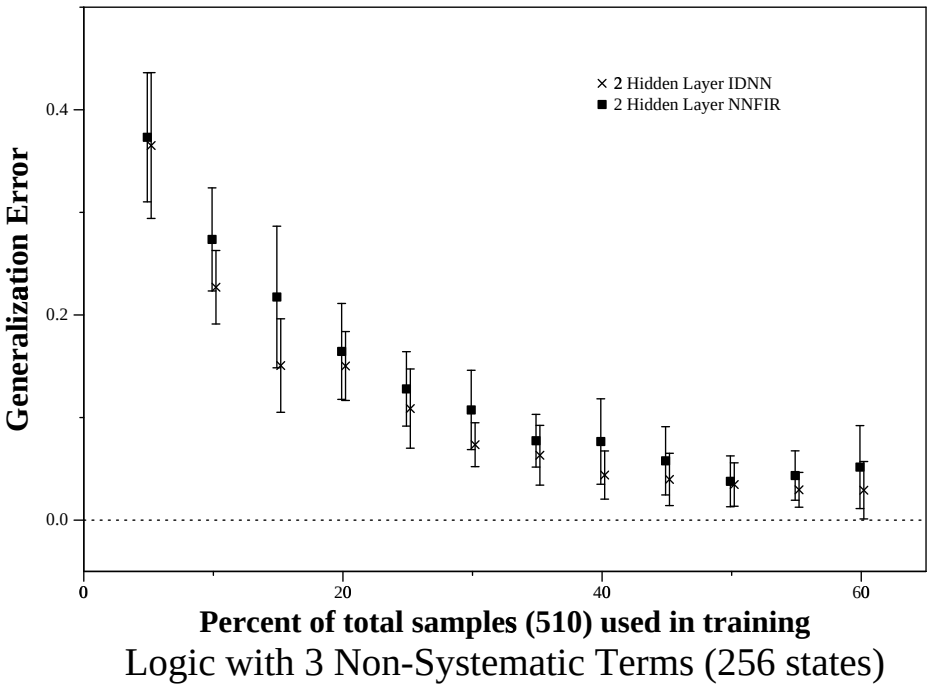

Figure 10: Generalization performance of a general NNFIR and an IDNN on a non-systematic problem

input window of any of the first layer feature detectors. To insure that the network had simultaneous access to information from an entire string of length seven, the maximum length of a string in the training set, we added one internal delay for each first layer hidden unit. The resulting architecture had two input delays, four internal delays after each first layer hidden unit, and a total of 57 weights.

Similarly, we shortened the input tapped delay line on the IDNN architecture. Since the entire mapping function contains only seven contiguous variables, the original IDNN architecture contained a final delay element that would never be used. The resulting IDNN architecture had six input delays. We added a single node in the first hidden layer to bring the total number of weights in the network to 59.

Note that each term in the mapping function covers a range of input items of width four, whereas the first layer nodes of the NNFIR architecture see only three input items at a time. Thus the input window of the NNFIR architecture does not cover an entire term.

Figure 11 shows the performance of the two architectures on this problem. The graph shows that the IDNN network does better than the NNFIR with small training sets, but the NNFIR architecture does better with large training sets. The difference between the two curves is not significant ($MS_{arch} = 0.0065$, $MS_{error} = 0.0028$, $F(1,456) = 2.343$, $p = 0.126$), but the interaction between the architecture factor and the training set size factor is significant ($MS_{arch \times size} = 0.0059$, $F(11,456) = 2.130$, $p = 0.017$).

Note that the NNFIR architecture does not perform as well in this experiment as it did

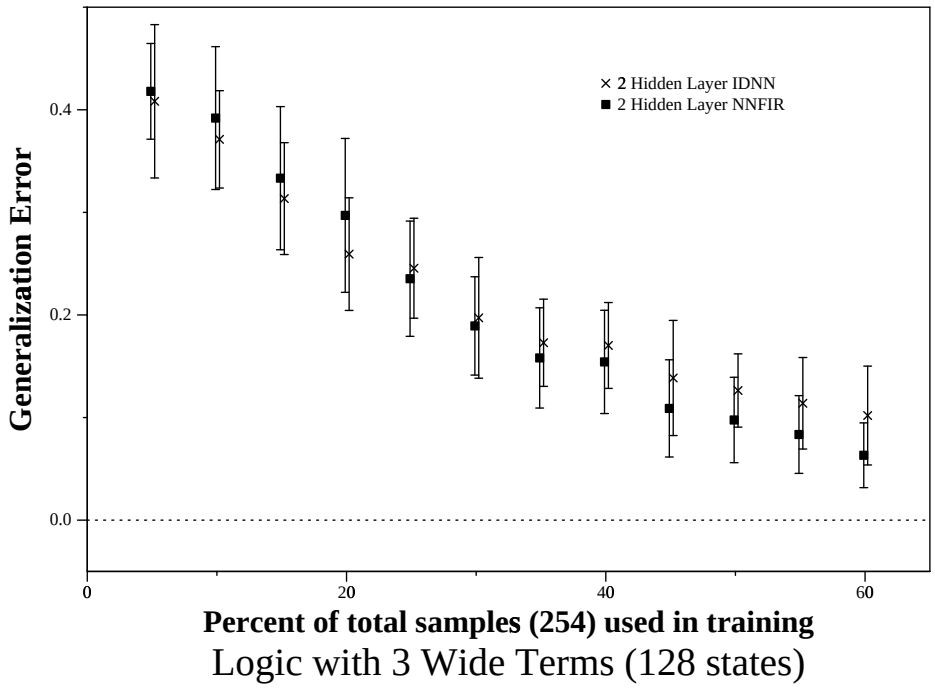

Figure 11: Generalization performance of a general NNFIR and an IDNN on a wide problem

in the earlier experiment with four repetitive terms in which the input window was large enough to cover an entire term (figure 9). If window size had no effect on performance we would expect the problem with fewer terms to perform better. Instead we see poorer performance in the problem with fewer terms. This is most likely due to the fact that the size of a term is wider than the NNFIR input window. The decrement in performance along with the similarity in performance of the NNFIR and IDNN networks on this task implies that the NNFIR architecture is indeed sensitive to the input window width.

8 Conclusions

This work serves as a foil to some potential misconceptions concerning language induction using neural networks. First, we have precisely defined the subclass of finite state machines which can be induced by the NNFIR architecture. This class is the definite memory machines. We have shown not only that the network architecture is capable of representing the languages in this class, but that NNFIRs are capable of learning languages of this class using the traditional back propagation learning algorithm.

Second, the work shows that some finite state machines can be learned using a feed-forward neural network. This is contrary to the normal practice of using recurrent networks in this role. The class of networks learnable using the NNFIR architecture is not limited to finite length strings, but includes machines whose transition diagrams have loops. This contradicts a natural intuition which equates loops in the FSM transition diagram with

recurrent loops in the network architecture.

Third, the work demonstrates that the number of states in an FSM is not necessarily a good predictor of the learnability of its accepted language. We are able to learn a 2048-state FSM with a simple feed-forward architecture using few training examples. This is possible because, we chose a language which can be represented in its sequential representation using a small amount of logic. In our example, the size of the logic function appears to have more influence than the number of states in determining the difficulty of the language to be learned.

Fourth, we investigated the utility of internal delay lines and found them useful in dealing with a language generated from combining fixed-length, repetitive constituents. We also found the NNFIR performance to degrade when those constituents exceeded the width of the input window.

A Minimality Theorem

Definition of n-group. An *n-group* is a set of all the 2^n nodes in a deBruijn graph whose names are all identical in the rightmost $d-n$ digits, where d is the number of digits in a node name.

Lemma 1 *If for each 1-group in the graph, one and only one of the pair of nodes in the group is an accepting state, then any two nodes, x and y, which are distinct members of an n-group, are distinguishable.*

Proof. We will prove this by induction on n.

Base case: $n = 1$. x and y are distinct members of a 1-group. The two states in a 1-group are distinguishable by the empty string since by the premise one is an accepting state and one is a rejecting state.

Induction Step. The induction hypothesis is that all $(n-1)$-groups are distinguishable. x and y are distinct members of an n-group. If x and y are also members of some $(n-1)$-group, then they are distinguishable and we are done with the proof, so assume x and y are not members of any $(n-1)$-group. Under these assumptions, the rightmost $d-n$ digits of their names are identical, but the next rightmost digit is not (or they would be members of an $(n-1)$-group). The names of x and y are, therefore, of the form $< x_1 \ldots x_{n-1} x_n x_{n+1} \ldots x_d >$ and $< y_1 \ldots y_{n-1} \overline{x}_n x_{n+1} \ldots x_d >$ respectively. The input string, 0, will take them respectively into states $< x_2 \ldots x_{n-1} x_n x_{n+1} \ldots x_d 0 >$ and $< y_2 \ldots y_{n-1} \overline{x}_n x_{n+1} \ldots x_d 0 >$ which are distinct members of a single $(n-1)$-group. By the induction hypothesis two members of an $(n-1)$-group are distinguishable. Therefore there exists some string ω that distinguishes them. The string 0ω will then distinguish between x and y. $\square$

Lemma 2 *If for each* 1*-group in the graph, one and only one of the pair of nodes in the group is an accepting state, then for every pair of distinct nodes in the graph, the two nodes are distinguishable.*

Proof. Let x and y be arbitrary nodes in the graph. Define g as the greatest integer such that the names of x and y share g rightmost identical digits. There are 2 possible cases.

Case 1: $g > 0$. Since the names of x and y share g rightmost digits, x and y are members of a $(d-g)$-group. By lemma 1 then, x and y are distinguishable.

Case 2: $g = 0$. Since $g = 0$, x and y have a distinct rightmost digits in their names. If the name of x is $< x_1x_2x_3 \ldots x_{d-1}x_d >$, then we know the name of y is of the form $< y_1y_2y_3 \ldots y_{d-1}\overline{x}_d >$ The input string 0^{d-1} will take x to the node named $< x_d000 \ldots 0 >$, and y to the node named $< \overline{x}_d000 \ldots 0 >$. These two destination nodes share $d-1$ rightmost digits (all 0s) and therefore are in the same 1-group. By the premise of this lemma, one of these nodes is an accepting state and one is a rejecting state. Therefore x and y are distinguishable by 0^{d-1}.

Theorem 1 *A necessary and sufficient condition for an assignment of accepting states to the nodes of a deBruijn graph of diameter* d *to result in a minimal FSM is that for each* 1*-group in the graph, one and only one of the pair of nodes in the group is an accepting state.*

Proof. We first show that if the graph is minimal, then one and only one of the pair of nodes in each 1-group is an accepting state. The fact that the graph is minimal implies that every pair of nodes in the graph is distinguishable. Specifically, the two nodes in a 1-group are distinguishable. By definition, a 1 input takes the two nodes in a 1-group to the same state, and likewise for a 0 input. For the two nodes to be distinguishable then, they must produce a distinguishable output on the null string. Therefore, one node must be an accepting state and the other a rejecting state.

Next we show that if for each 1-group in the graph, one and only one of the pair of nodes in the group is an accepting state, then the graph is minimal. It follows from lemma 2, that for every pair of distinct nodes in the graph, the two nodes are distinguishable. To prove minimality we also need to show that each node in the graph is reachable from the start node. By the definition of a deBruijn graph, it is clear that the string corresponding to the name of a node will cause a transition to that node from any node in the graph. Each node in the graph is therefore reachable, and every pair of nodes is distinguishable. Therefore, the FSM defined by this procedure is minimal. □

References

[Cleeremans et al., 1989] Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381.

[Day and Davenport, 1993] Day, S. and Davenport, M. (1993). Continuous-time temporal back-progagation with adaptive time delays. *IEEE Transactions on Neural Networks*, 4:348–354.

[Elman, 1991] Elman, J. L. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7(2/3):195–226.

[Giles et al., 1994] Giles, C. L., Horne, B. G., and Lin, T. (1994). Learning a class of large finite state machines with a recurrent neural network. Technical Report UMIACS-TR-94-94 and CS-TR-3328, Intitute for Advanced Computer Studies, Univerity of Maryland.

[Giles et al., 1992] Giles, C. L., Sun, G. Z., Chen, H. H., Lee, Y. C., and Chen, D. (1992). Higher order recurrent networks and grammatical inference. *Neural Computation*, 4(3):393–405.

[Golomb, 1982] Golomb, S. (1982). *Shift Register Sequences*. Aegean Park Press, Laguna Hills, CA.

[Kohavi, 1978] Kohavi, Z. (1978). *Switching and Finite Automata Theory*. McGraw-Hill, Inc., New York, NY, second edition.

[Krogh and Hertz, 1992] Krogh, A. and Hertz, J. (1992). A simple weight decay can improve generalization. In [Moody et al., 1992], pages 950–957.

[Lang et al., 1990] Lang, K., Waibel, A., and Hinton, G. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3(1):23–44.

[Lin et al., 1992] Lin, D., Dayhoff, J., and Ligomenides, P. (1992). A learning algorithm for adaptive time-delays in a temporal neural network. Technical Report TR 92-59, Systems Research Center, University of Maryland, College Park, Maryland.

[Moody et al., 1992] Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors (1992). *Advances in Neural Information Processing Systems 4 Proceedings of the 1991 Conference*, Denver, Colorado. Morgan Kaufmann Publishers : San Francisco, CA.

[Pearlmutter, 1989] Pearlmutter, B. A. (1989). Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269.

[Pollack, 1991] Pollack, J. B. (1991). The induction of dynamical recognizers. *Machine Learning*, 7(2/3):227–252.

[Rice, 1988] Rice, J. A. (1988). *Mathematical Statistics and Data Analysis*. Brooks/Cole Publishing Company, Monterey, California.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, chapter 8. MIT Press, Cambridge, Mass.

[Waibel et al., 1989] Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. (1989). Phoneme recognition using time–delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(3):328–339.

[Wan, 1993] Wan, E. A. (1993). Time series prediction by using a connectionist network with internal delay lines. In Weigend, A. S. and Gershenfeld, N. A., editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*. Addison Wesley.

[Watrous and Kuhn, 1992] Watrous, R. and Kuhn, G. (1992). Induction of finite state languages using second-order recurrent networks. In [Moody et al., 1992], pages 309–316.