Time-Delay Neural Networks: Representation and Induction of Finite State Machines

Daniel S. Clouse, C. Lee Giles, Bill G. Horne, and Garrison W. Cottrell

D. S. Clouse and G. W. Cottrell are with the University of California, San Diego. E-mail: {dclouse,gcottrell}@ucsd.edu

C. L. Giles is with NEC Research Institute, Princeton, NJ 08540, and Institute for Advanced Computer Studies, U. of Maryland, College Park, Md. 20742. E-mail: giles@research.nj.nec.com

B. G. Horne is with NEC Research Institute, Princeton, NJ 08540. E-mail: horne@research.nj.nec.com

April 4, 1997 Accepted in IEEE Transactions on Neural Networks

Abstract

In this work, we characterize and contrast the capabilities of the general class of *time-delay neural networks* (TDNN), with *input delay neural networks* (IDNN), the subclass of TDNNs with delays limited to the inputs. Each class of networks is capable of representing the same set of languages, those embodied by the *definite memory machines* (DMM), a subclass of *finite state machines*. We demonstrate the close affinity between TDNNs and DMM languages by learning a very large DMM (2048 states) using only a few training examples. Even though both architectures are capable of representing the same class of languages, they have distinguishable learning biases. Intuition suggests that general TDNNs which include delays in hidden layers should perform well, compared to IDNNs, on problems in which the output can be expressed as a function on narrow input windows which repeat in time. On the other hand, these general TDNNs should perform poorly when the input windows are wide, or there is little repetition. We confirm these hypotheses via a set of simulations and statistical analysis.

Keywords

language induction, time-delay neural network, finite state machines, inductive bias, sequential circuits, temporal sequences, gradient-descent training, automata, memory, tap-delay lines

I. The TDNN Architecture and Its Relationship to Automata

In this paper we consider the class of neural network architectures called *time-delay neural networks* (TDNN) [1] [2], also known as the *neural network finite impulse response* (NNFIR) architecture [3]. This architecture has been used quite successfully in a number of practical applications including speech recognition [1], and time series prediction [3]. The purpose of this paper is to characterize the capabilities of this architecture in a simple domain and to contrast the learning bias of two subclasses of the architecture.

A TDNN is similar to a multi-layer perceptron in that all connections feed forward. The difference is that with the TDNN, the inputs to any node i can consist of the outputs of earlier nodes not only during the current time step t, but during some number, d, of previous time steps $(t-1, t-2, \dots t-d)$ as well. This is generally implemented using tap-delay lines. The activation function for node i at time t in such a network is given by equation 1:

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

where y_i^t is the output of node i at time t, w_{ijk} is the connection strength to node i from the output of node j at time $t-k$, and h is the activation function.

A natural restriction of the general TDNN topology is the class of TDNN architectures which have delays only on the input units. We call these *input delayed neural networks* (IDNNs). Since the class of IDNNs is a subclass of TDNNs, every IDNN is also a TDNN. Clearly then, the set of functions computable by the TDNNs 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 TDNN class [3][1].

We are interested in the problem of language induction because it has a rich theoretical background (see [4] for an overview, and the recent conferences [5] [6]), and it can serve as a test bed for evaluating and contrasting properties of particular neural network architectures. The language induction problem can be defined as follows: Given a set of binary strings, labeled as to which are included in a particular language L and which are not, the task of the network is to learn to correctly classify new strings which were not part of the original training set as to whether or not they are included in L. The language induction problem is a discrete domain, in that the network inputs are all taken from a discrete alphabet (e.g. $\{0,1\}$), and the outputs are interpreted as discrete.

The set of languages which can be representable by some TDNN are exactly those languages representable by a *definite memory machine* (DMM). Kohavi [7] (chapter 14) defines a DMM of order d as a *finite state machine* (FSM) whose present state can always be determined uniquely from the knowledge of the last d inputs. It is easy to show that the following definition is equivalent: a DMM of order d is a FSM whose accepting/rejecting behavior is a function of only the most recent d inputs.

Consider an IDNN which has a single output unit, and $d-1$ delays on its only input. The IDNN computes a function of the most recent d inputs (including the current input), therefore the IDNN accepts a DMM language. Since any boolean function can be repre-

[1]Wan (1993) suggests a method of transforming any TDNN into an IDNN which performs the same mapping. This transformation works for any TDNN, as defined above, for input strings which are longer than the total delay length of the network. It does not take into account the effect of the initial state of the networks as contained in the tap-delay memories, so the general TDNN architecture may, in fact, be able to represent languages not representable by any IDNN, if the internal delays of the TDNN are allowed to take on initial values which cannot be computed by any legal input sequence.

sented by a feed-forward network with enough hidden units, some IDNN exists which can perform any mapping from recent input history to any boolean discrimination function [8]. Therefore, the IDNNs can, in fact, represent any DMM. We have already seen that the TDNN and IDNN classes are essentially functionally equivalent. This implies that TDNNs implement DMMs as well.

The entire state of an IDNN is contained in the most recent d inputs. From any state the next state is completely determined by the shift-register behavior of the tap-delay line. The resulting transition diagram is called a directed de Bruijn graph of diameter d [9]. The task of the IDNN is to learn which states in this fixed transition diagram are accepting states and which rejecting.

We have determined the conditions under which such a transition diagram results in a minimal FSM. It is possible to change the contents of a shift register of length d to any desired value by shifting in the desired d bits. Likewise, in a de Bruijn graph of diameter d, for each state there exists some unique string of d inputs which will lead to that state no matter what the starting state. We label each state with its unique string of d inputs. In our labeling scheme, the leftmost bit is the first bit presented. A necessary and sufficient condition for an assignment of accepting states to the nodes of a de Bruijn graph of diameter d to result in a minimal FSM is that for each pair of states whose labels share all but the leftmost bit, one and only one of the pair is an accepting state. A proof is presented in the appendix.

II. Learning a large DMM

An interesting demonstration of this understanding of TDNNs and DMMs is to learn a DMM with many states using a small subset of the possible training examples. This section is similar in spirit to Giles, Horne, and Lin [10] in which the authors show that a discrete time recurrent neural network is capable of learning a large finite memory machine (FMM), a larger subset of the FSMs.

The machine learned here is a DMM of order 11. It is defined by the function given in equation 2 which maps from recent inputs (or state labels) to "accept" (1) or "reject" (0). Here, the symbol $\overline{\oplus}$ represents the *XNOR* function (i.e. $x \overline{\oplus} y = xy + \bar{x}\bar{y}$). Also, u_k represents the input at time k, and y_k represents the output of the function at time k.

The overbar notation, $\overline{u}_k$, represents the negation of u_k.

$$y_k = u_{k-10}\overline{\oplus}\left(\overline{u}_k\overline{u}_{k-1}\overline{u}_{k-2} + \overline{u}_{k-2}u_{k-3} + u_{k-1}u_{k-2}\right) \tag{2}$$

This logic function was chosen so that the minimal FSM representation would be a de Bruijn graph of diameter 11, requiring $2^{11} = 2048$ states.

The network architecture consists of an IDNN with 10 tap-delays on the input, seven hidden nodes and a single sigmoidal output node. To create training and test sets, we generated all strings of length 1 to 11, 4094 in all, and labeled them according to the DMM language. A trial consisted of a randomly chosen percentage of these strings on which the network was trained to the error criterion described below.

During training (and testing), before introduction of a new string, the values of all the tap-delays were set to 0. Online back propagation [11] with a learning rate of 0.25 and momentum of 0.25 was used for training. 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[2]. This effectively speeds up the algorithm by avoiding gradient calculations for any string which will result in very small weight adjustments. To avoid over-training, and improve generalization, 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. This particular training method was chosen because we have found it to be effective on a number of similar problems. All learning algorithm parameters were set independently of this particular problem.

Figure 1 plots the average classification error on the test set for various sizes of training sets. 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 or equal to 0.5. Therefore, a value of 0.5 on the graph indicates chance performance. The values plotted

[2]This is equivalent to minimizing the following cost function, where y_i is the output of node i and t_i is the corresponding target:

$$\sum_i g(t_i - y_i) \quad \text{where} \quad g(x) = \begin{cases} x^2 & \text{if } |x| > 0.2 \\ 0 & \text{otherwise} \end{cases}$$

This cost function is consistent with our task in which an output greater than 0.5 is considered a 1, and any output less than 0.5 is considered a 0.

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. To our knowledge this is the first demonstration of induction of such a large FSM using a neural network. This is possible because of the strong representational bias inherent in the IDNN architecture. The logic required to implement the mapping function is actually quite small; many of the delayed inputs are not even used! In this sense one could interpret these results as best-case.

III. Characterizing the bias of IDNNs and HDNNs

Earlier we stated that the IDNN architecture and the general TDNN 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 section, 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 an IDNN and a comparably-sized TDNN network which includes delays on the hidden layer. We refer to such a network as a *hidden delayed neural network* or HDNN.

Each of the architectures we use in the following simulations contains a single input and a single output. The IDNN architecture contains eight input delays, and two hidden layers, one with four units, and the other with five. The HDNN architecture contains three input delays and two hidden layers, each with only three nodes. In addition, the outputs of the first hidden layer of the HDNN each have five delays. Note that in each architecture, the output node computes a function of the current input along with eight input delays. Also, each architecture was designed to have four layers, and approximately the same number of weights. Counting biases, the HDNN architecture has 76 weights, and the IDNN architecture has 79.

Our intuition suggests that the HDNN architecture may be better at learning DMM problems in which the logic function which defines the DMM contains identical terms which repeat across time. This intuition comes from viewing each node in the first hidden layer of the HDNN architecture as developing a feature detector which looks for a specific term in its input window. If a term repeats, a single feature detector can be used to recognize it each time. Since the outputs of the first hidden layer feed into tap-delay lines, later layers have access to the findings of the feature detectors at several times steps, and can combine these findings into the final output. The IDNN architecture, lacking the internal tap-delay lines should find it more difficult to pick up on this kind of regularity in the input.

If our intuition is correct, we should see the expected advantage that the HDNN architecture has on problems with narrow, repeated terms disappear when the terms become wider than the feature detectors, or when there is little repetition in the terms. The IDNN architecture, on the other hand, should have comparatively little problem with these kinds of problems.

To test these hypothesis, we ran a series of simulations which contrast the performance of an IDNN network with a HDNN network in learning four DMMs each defined by a particular mapping function. We call the first DMM, the *narrow-repeated* problem because the mapping function which defines it consists of the same narrow term four times, shifted in time. Each term is narrow enough to fit inside the input window of the HDNN architecture. The mapping function for the narrow-repeated problem is given in equation 3.

$$\begin{aligned} y_k = u_{k-8}\overline{\oplus}(\quad & u_k u_{k-2}\overline{u}_{k-3} + u_{k-1}u_{k-3}\overline{u}_{k-4} + \\ & u_{k-3}u_{k-5}\overline{u}_{k-6} + u_{k-4}u_{k-6}\overline{u}_{k-7}) \end{aligned} \tag{3}$$

Like all problems in this paper, this function has been chosen so that the minimal FSM maps directly onto a de Bruijn graph. All the problems in this section require 512 states. 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 HDNN architectures. This is also true for two of the other problems in this section. We have not been able to find a set of weights for the HDNN architecture for the wide-unrepeated term problem, defined below.

The *narrow-unrepeated* problem is narrow, in that each term in the function that defines it is capable of being seen in a single time step by the HDNN architecture. We call the problem unrepeated because no term is simply a shifted representation of any other term. The mapping function for the narrow-unrepeated problem is given in equation 4.

$$\begin{aligned} y_k = u_{k-8}\overline{\oplus}(& u_k u_{k-2}\overline{u}_{k-3} + \overline{u}_{k-1}u_{k-3}u_{k-4} + \\ & u_{k-3}\overline{u}_{k-5}u_{k-6} + \overline{u}_{k-4}\overline{u}_{k-6}\overline{u}_{k-7}) \end{aligned} \tag{4}$$

Similarly, equations 5 and 6 give the mapping functions, respectively, for the *wide-repeated* and *wide-unrepeated* DMMs. These two equations are identical to their narrow counterparts, except for the addition of an extra space in each term.

$$\begin{aligned} y_k = u_{k-8}\overline{\oplus}(& u_k u_{k-2}\overline{u}_{k-4} + u_{k-1}u_{k-3}\overline{u}_{k-5} + \\ & u_{k-2}u_{k-4}\overline{u}_{k-6} + u_{k-3}u_{k-5}\overline{u}_{k-7}) oldEquation5 \end{aligned} \tag{5}$$

$$\begin{aligned} y_k = u_{k-8}\overline{\oplus}(& u_k u_{k-3}\overline{u}_{k-4} + \overline{u}_{k-1}u_{k-4}u_{k-5} + \\ & u_{k-2}\overline{u}_{k-5}u_{k-6} + \overline{u}_{k-3}\overline{u}_{k-6}\overline{u}_{k-7}) \end{aligned} \tag{6}$$

The training method used for the IDNN network was identical to the simulation presented earlier except training was stopped after 8000 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 TDNN architecture is slightly more complicated. Error from the final output must be propagated backwards across the internal tap-delay lines. See [3] for a description of the general algorithm. With the exception of the difference of this slightly more complicated training method, all details of the HDNN training were identical to that of the IDNN training.

The results of the simulations for the four problems are presented in figure 2. One graph is presented for each of the four problems. Each graph plots the generalization error for each of the two architectures. Plotted points are the mean classification error on the test set for the IDNN and HDNN architectures averaged across 20 trials at each training set size.

If our hypotheses are correct, we would expect the effect of wider terms to be strong for the performance of the HDNN, and weak for the IDNN. Likewise, we would expect the effect of repeated terms to be more pronounced for HDNNs. Figure 2 supports these expectations. Comparing each of the wide curves to their respective narrow curves, the position of the IDNN curve changes little between graphs, while the position of the HDNN curve is quite dynamic. This same effect is also noticeable, though not as pronounced, when comparing repeated curves to their respective unrepeated curves.

Statistical tests also support our expectations. We ran a multi-factor analysis of variance (ANOVA) [12] on the data summarized in the graphs. The factors of interest were the architecture (HDNN or IDNN), the width of terms (narrow or wide), the uniformity of terms (repeated or unrepeated), and the percent of possible patterns which were used in training. It should come as no surprise that the results indicate that each of the factors individually has a significant effect on generalization ($MS_{error} = 0.0015$, $MS_{arch} = 0.1685$, $F(1, 1824) = 115.1$; $MS_{width} = 0.4396$, $F(1, 1824) = 300.4$; $MS_{unif} = 0.3219$, $F(1, 1824) = 220.0$; $MS_{percent} = 1.840$, $F(11, 1824) = 1257$). All individual factors are significant at $p < 0.001$[3] More to the point, the test also indicates that there is a significant interaction between the architecture and width factors ($MS_{arch \times width} = 0.3430$, $F(1, 1824) = 234.4$), and between the architecture and uniformity factors ($MS_{arch \times unif} = 0.1181$, $F(1, 1824) = 80.69$). These two interactions are also significant at $p < 0.001$, indicating that width and uniformity each has a stronger effect on the HDNN than on the IDNN.

[3] For each ANOVA test, we compare two estimates of the variance of the data. One estimate, based on MS_{error} is the same for each test, and is independent of the null hypothesis. The other estimate of variance is dependent on the null hypothesis and thus changes depending on what hypothesis we are testing. For example, MS_{arch} is dependent on the choice of architectures, and therefore is used to test the null hypothesis that there is no effect of the difference in architectures on the error rate. Similarly, MS_{width}, MS_{unif}, and $MSpercent$ are dependent respectively on the width of terms, whether repeated or unrepeated terms were used, and on the percentage of patterns used in training. In each test, if the null hypothesis is true, then F, the ratio between the dependent MS and $MSerror$, is expected to be close to 1. As it stands, each of our F values 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, in each case, this probability is small, we are justified in rejecting the null hypothesis.

We have shown that the HDNN architecture is more affected by variation in term width and term repetition than the IDNN architecture. We can also ask which architecture performs better on the various problems? We note that the HDNN curve is clearly lower for the narrow-repeated problem, and also appears to be lower on average for the narrow-unrepeated problem. The IDNN curve appears lower for the wide-unrepeated problem, while the wide-repeated problem is difficult to distinguish. Statistical analysis is consistent with this observation. In further planned tests, for each problem individually we compared the mean performance of the HDNN architecture across the entire curve to that of the IDNN architecture. The HDNN performs significantly better than the IDNN in the narrow-repeated problem ($MS_{error} = 0.0015$, $MS_{arch} = 0.5400$, $F(1, 1824) = 369.0$), and in the narrow-unrepeated problem ($MS_{arch} = 0.0683$, $F(1, 1824) = 46.7$). The IDNN performs significantly better in the wide-unrepeated problem ($MS_{arch} = 0.0378$, $F(1, 1824) = 25.83$). All of these comparisons are significant at $p < 0.001$. The mean performance for the HDNN and IDNN architectures on the wide-repeated problem are almost identical, and the one-way ANOVA test fails to reject the null hypothesis that the two means are identical ($MS_{arch} = 0.0004$, $F(1, 1824) = 0.273$, $p > 0.60$)[4]. So the HDNN architecture generalizes better on both the narrow problems, while the IDNN architecture achieves lower error only on the wide-unrepeated problems.

This experiment serves as confirmation for our intuition: Delays on the hidden layers are useful in problems composed of narrow, repeated features. When it is known that a specific problem is composed of narrow, repeated features, this detailed knowledge can be used to bias the network architecture to generalize well on that specific problem. When the features are wide or their width is not known, and in situations where repeated features are unlikely, an architecture without hidden delays may outperform an architecture which includes them.

The discrete domain in which we have been working has allowed us to generate a small set of related problems which demonstrate the advantages and disadvantages of delays

[4]Though the presence of an effect of architecture on the wide-repeated problem is not statistically significant, the interaction between architecture and the size of training set does show a significant effect ($MS_{arch \times percent} = 0.0069$, $F(11, 1824) = 4.6$, $p < 0.001$). This suggests that the two curves were indeed chosen from different distributions. Nevertheless the difference in mean performance is not different enough as to be statistically significant.

on the hidden units. We have not directly addressed the question of whether this effect extends to the more general, real-valued domain, but the results are highly suggestive.

IV. 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 represented by the TDNN 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 TDNNs 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 TDNN architecture is not limited to finite length strings, but includes machines whose transition diagrams have loops.

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 using a small amount of logic. In our example, the complexity 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 present a series of simulations that highlight the difference between *representational bias*, and *learning bias*. Two subclasses of TDNNs, those with and without delays on the outputs of hidden units, are both capable of *representing* essentially the same class of problems, but each is well-suited to *learning* a different set of problems. The HDNN, which includes delays on the hidden units, has an advantage on problems which are composed of narrow, repeated elements, while the IDNN generalizes better when this is not the case.

V. Acknowledgments

The authors would like to thank a number of anonymous reviewers for helpful comments. Also, thanks to T. Lin, and to the GEURU research group for useful discussions. Daniel

Clouse was supported in part by a USPHS Predoctoral Traineeship.

References

[1] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang, "Phoneme recognition using time–delay neural networks," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37, no. 3, pp. 328–339, 1989.

[2] K. Lang, A. Waibel, and G. Hinton, "A time-delay neural network architecture for isolated word recognition," *Neural Networks*, vol. 3, no. 1, pp. 23–44, 1990.

[3] Eric A. Wan, "Time series prediction by using a connectionist network with internal delay lines," in *Time Series Prediction: Forecasting the Future and Understanding the Past*, A. S. Weigend and N. A. Gershenfeld, Eds. Addison Wesley, 1993.

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

[5] Rafael C. Carrasco and Jose Oncina, *Grammatical Inference and Applications. Proceedings of the 2nd International Colloquium*, vol. 862 of *Lecture Notes on Computer Science*, Springer-Verlag, 1994.

[6] Laurent Miclet and Colin de la Higuera, *Grammatical Inference: Learning Syntax from Sentences. Proceedings of the 3nd International Colloquium*, vol. 1147 of *Lecture Notes on Computer Science*, Springer-Verlag, 1996.

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

[8] B. G. Horne and D. R. Hush, "On the node complexity of neural networks," *Neural Networks*, vol. 7, no. 9, pp. 1413–1426, 1994.

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

[10] C. Lee Giles, Bill G. Horne, and T. Lin, "Learning a class of large finite state machines with a recurrent neural network," *Neural Networks*, vol. 8, no. 9, pp. 1359–1365, 1995.

[11] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, David E. Rumelhart and James L. McClelland, Eds., vol. 1, chapter 8. MIT Press, Cambridge, Mass., 1986.

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

Appendix

I. Minimality Theorem

Definition of similar pair. A *similar pair* is a pair of distinct nodes in a de Bruijn graph whose labels are identical in the rightmost $d-1$ digits, where d is the diameter of the de Bruijn graph.

Lemma 1: If for each similar pair in the graph, one and only one node of the pair is an accepting state, then every pair of distinct nodes in the graph is distinguishable.

Proof. Let x and y be arbitrary, distinct nodes in the graph. Define g as the greatest integer such that the labels of x and y share g rightmost identical digits. If the label of x is of the form $< x_d x_{d-1} \ldots x_{g+2} x_{g+1} x_g \ldots x_2 x_1 >$, then the label of y must be of the form

$< y_d y_{d-1} \ldots y_{g+2} \overline{x}_{g+1} x_g \ldots x_2 x_1 >$. Since x and y are distinct, $g < d$, and so the $g+1$th digit from the right must be opposites in these two labels. The input string 0^{g-1} will take x to $< x_{g+1} x_g \ldots x_2 x_1 0^{g-1} >$, and y to $< \overline{x}_{g+1} x_g \ldots x_2 x_1 0^{g-1} >$. These two destination nodes share $d-1$ rightmost digits and therefore are a similar pair. 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^{g-1}.

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

Proof. We first show that if the graph is minimal, then one and only one of the nodes in each similar pair 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 similar pair are distinguishable. By definition, a 1 input takes the two nodes in a similar pair 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 similar pair in the graph, one and only one of the pair is an accepting state, then the graph is minimal. It follows from the lemma, above, 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 de Bruijn graph, it is clear that the string corresponding to the label 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. □

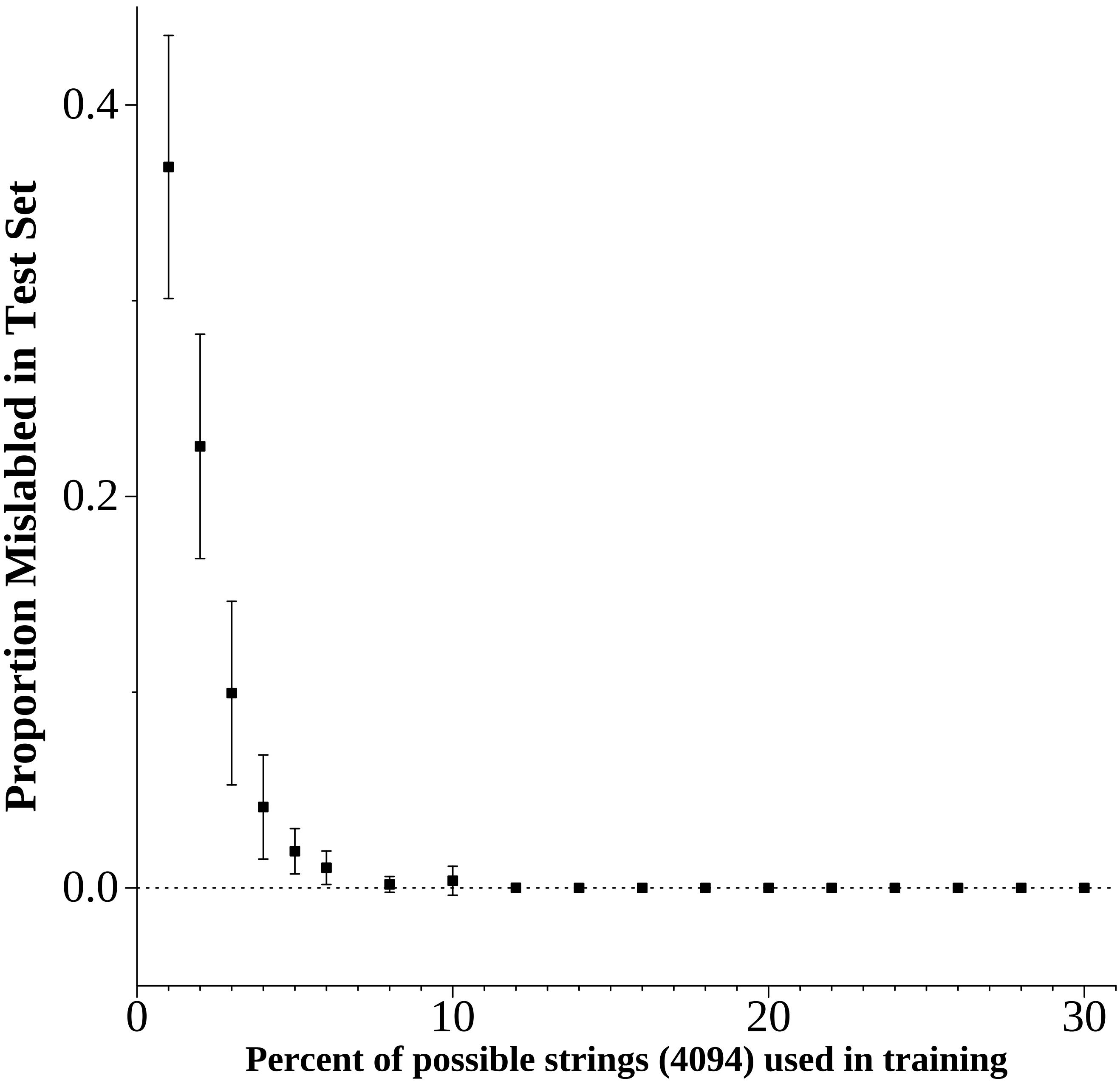

Fig. 1. Generalization as a function of training set size on the 2048 state DMM.

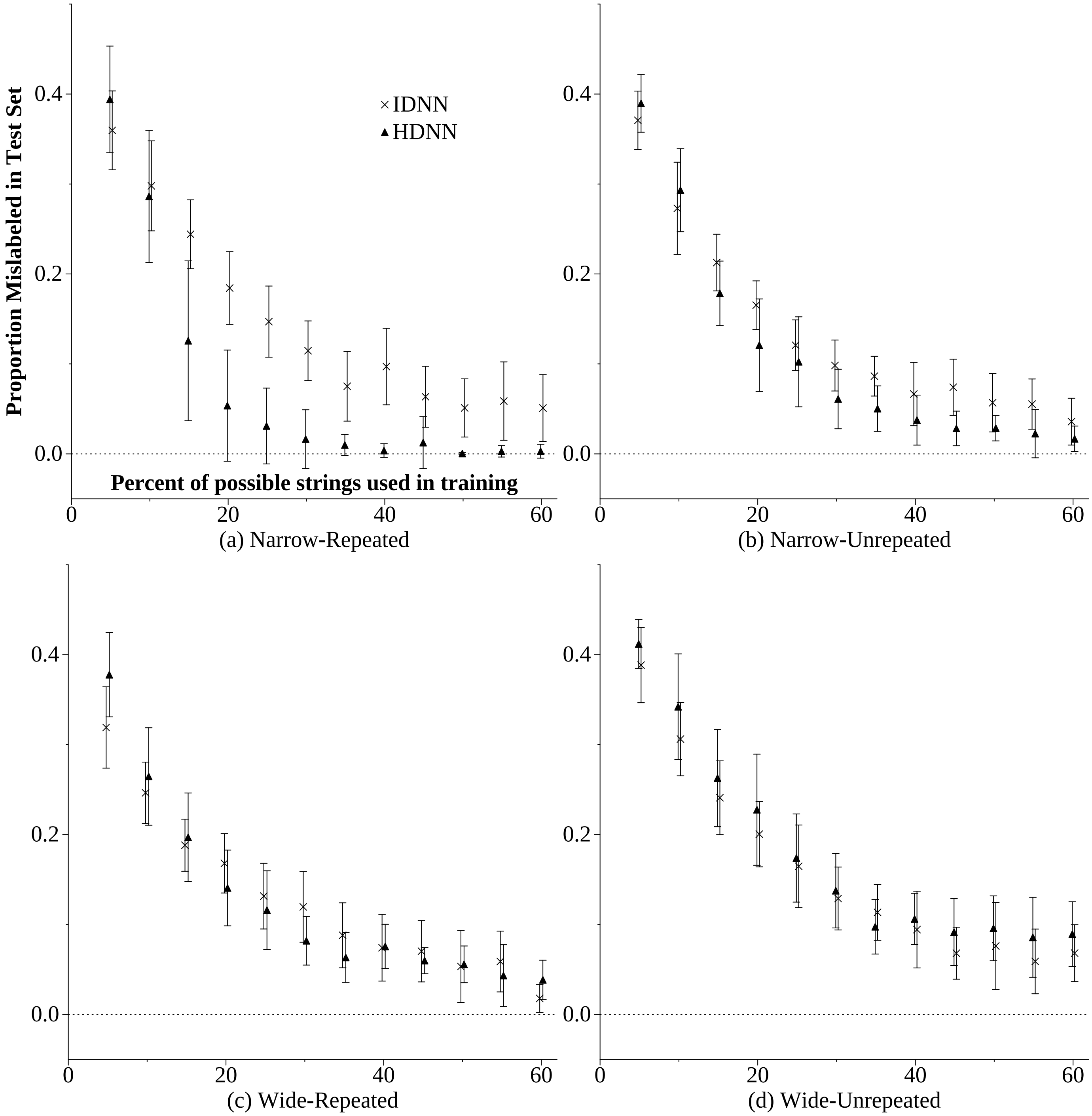

Fig. 2. Generalization of an IDNN and an HDNN on four DMM problems