Discrete Recurrent Neural Networks for Grammatical Inference

Zheng Zeng, *Student Member, IEEE,* Rodney M. Goodman, *Member, IEEE,* and Padhraic Smyth, *Member, IEEE*

***Abstract*—We describe a novel neural architecture for learning deterministic context-free grammars, or equivalently, deterministic pushdown automata. The unique feature of the proposed network is that it forms stable state representations during learning—previous work has shown that conventional analog recurrent networks can be inherently unstable in that they cannot retain their state memory for long input strings. We have recently introduced the discrete recurrent network architecture for learning finite-state automata. Here we extend this model to include a discrete external stack with discrete symbols. A composite error function is described to handle the different situations encountered in learning. The pseudo-gradient learning method (introduced in previous work) is in turn extended for the minimization of these error functions. Empirical trials validating the effectiveness of the pseudo-gradient learning method are presented, for networks both with and without an external stack. Experimental results show that the new networks are successful in learning some simple pushdown automata, though overfitting and non-convergent learning can also occur. Once learned, the internal representation of the network is provably stable; i.e., it classifies unseen strings of arbitrary length with 100% accuracy.**

I. Introduction

RECURRENT neural networks have recently been demonstrated to have the ability to learn simple grammars [2], [4], [5], [6], [8], [11], [12], [13], [17] and to learn deterministic context-free grammars by using an external "continuous stack" [3]. In this paper we focus our attention on "second-order" recurrent network structure of the type proposed by Giles *et al.* in [8]—henceforth, this particular model is referred to as the analog second-order network. A typical analog second-order network is shown in Fig. 1. These higher-order networks are particularly adept at learning grammars when compared to the simple recurrent network structure (also known as the Elman structures [4], [5]) that do not use product units [8].

In previous work [18], we have found that in learning a regular grammar the analog second-order network attempts to form clusters of points in hidden unit activation space as its representation of the states of the grammar. Once formed, these clusters are stable for short strings (strings with lengths not much longer than the maximum length of training strings) in the sense that the hidden unit activations move from one distinct cluster to another as the network follows a trajectory in hidden unit space. However, for most of the trained networks, when sufficiently long strings are presented for testing, the hidden unit activations start to converge to a single cluster and the original clusters ultimately become indistinguishable [18]. Similar behavior for recurrent networks with different structures has been reported elsewhere [12], [14]. In order to achieve stability for long strings, we proposed in [18] a discrete recurrent network architecture that uses discretization in its feedback links. A pseudo-gradient training method is used to train the network. Note that an alternative approach is to use a conventional analog network in training and to then apply various clustering techniques in the hidden unit activation space (after learning) to enforce stability [8]. While this is a valid approach, here we are more interested in constructing a network that stabilizes itself (or equivalently, automatically performs the clustering) *during* the learning process.

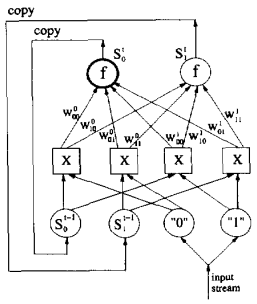


Fig. 1. An analog second-order network. Each square unit takes the product of its two inputs as its output. When the current input is 1, input unit "0" has value 0, and input unit "1" has value 1, and vice versa. The thickly circled unit is the indicator unit, whose desired value is close to 1 when the input is legal and close to 0 when the input is illegal.

In this paper, for context-free grammars, we introduce a discrete network architecture that has an external discrete stack. In the proposed network, instead of clusters, the states of the network consist of isolated points in hidden-unit activation space. Hence, once formed, the internal state representation is stable in a manner independent of string length.

The remaining part of the paper is organized as follows: Section II summarizes our previous work on discrete recurrent networks for learning simple grammars. Section III introduces discrete recurrent networks that use external stacks, and the pseudo-gradient training algorithm necessary for such models. Section IV presents experimental results in learning deterministic context-free grammars using the discrete stack model, and Section V concludes the paper.

Manuscript revised September 23, 1993. The research described in this paper was supported in part by ONR andARPA under grant number N00014-92-J-1860. In addition this work was carried out in part by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Z. Zeng and R. M. Goodman are with the Department of Electrical Engineering, California Institute of Technology, Pasadena, CA 91125.

P. Smyth is with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109.

IEEE Log Number 9214805.

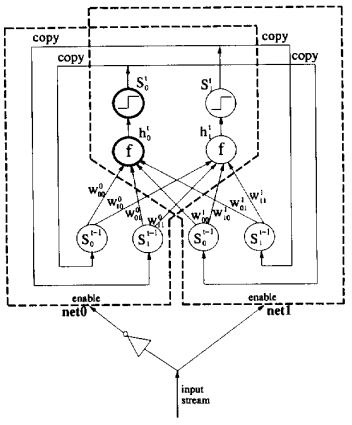


Fig. 2. A discretized second-order network. The thick circled unith_0^t is the indicator unit: $h_0^t > 0.5$ for legal strings and $S_0^t < 0.5$ for illegal strings.

II. Summary of Previous Work on Discrete Recurrent Networks for Learning Simple Grammars

A. Basic Structure of Discrete Recurrent Networks

A discrete recurrent network can be constructed by simply taking an analog recurrent network and adding threshold units to all the feedback links. In the case of second-order networks, one can represent the structure as two separate networks controlled by a gating switch (Fig. 2) as follows [18]: The network consists of two first-order networks with shared hidden units. The common hidden unit values are discretized and copied back to both **net0** and **net1** after each time step, and the input stream acts like a switching control to enable or disable one of the two "subnetworks." For example, when the current input is 0, **net0** is enabled while **net1** is disabled. The hidden unit values are then decided by the hidden unit values from the previous time step weighted by the weights in **net0**. The hidden unit activation function is the standard sigmoid function, $f(x) = \frac{1}{1+e^{-x}}$. The discretization function is defined to be:

$$D(x) = \begin{cases} 0.8 & \text{if } x \geq 0.5 \\ 0.2 & \text{if } x < 0.5. \end{cases} \tag{1}$$

The values 0.2 and 0.8 are chosen instead of 0 and 1 here in order to give some power of influence to each of the current hidden unit values over the next time step. A unit with value 0 would eliminate any influence of that unit over the next time step. This is the general network structure that was used in our first set of experiments [18].

We use h_i^t to denote the analog value of hidden unit i at time step t, and S_i^t to denote the discretized value of hidden unit i at time step t. w_{ij}^n is the weight from layer 1, unit j to layer 2, unit i in **net** n. $n = 0$ or 1 in the case of binary inputs. Hidden unit h_0^t is chosen to be a special indicator unit whose activation should be greater than 0.5 at the end of a legal string, or smaller than 0.5 otherwise. At time $t = 0$, initialize S_0^0 to be 0.8 and all other S_i^0's to be 0.2; i.e., assume that the null string is a legal string. The network weights are initialized randomly with a uniform distribution between −1 and 1.

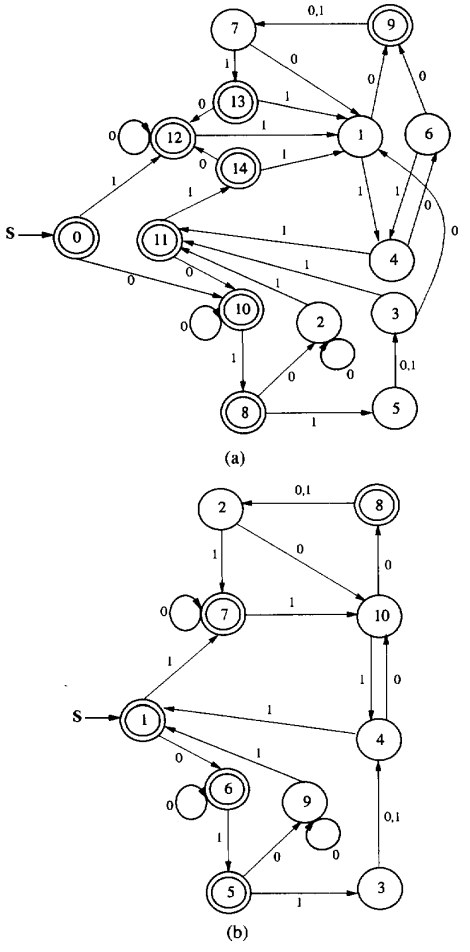


Fig. 3. Extracted-state machine from the discretized network after learning the 10–state machine: (a) 15–state machine extracted directly from the discrete activation space, (b) equivalent minimal 10–state machine of (a). Note that the state structure in (a) and (b) are quite similar; for example, states 1 and 6 in (a) are equivalent to 10 in (b), and states 12, 13, and 14 in (a) play a similar role to state 7 in (b).

In [18] we showed that discretization can be included during *both* training and testing using the formulae below. Note that from the formulae one can clearly see that in operational mode, i.e., when *testing*, the network is equivalent to a network with discretization only:

$$h_i^t = f(\sum_j w_{ij}^{x^t} S_j^{t-1}),\ \forall i, t,$$

$$S_i^t = D(h_i^t),$$

$$\text{where } D(x) = \begin{cases} 0.8 & \text{if } x \geq 0.5 \\ 0.2 & \text{if } x < 0.5, \end{cases}$$

$$\Rightarrow S_i^t = D(f(\sum_j w_{ij}^{x^t} S_j^{t-1}))$$

$$\equiv D_0(\sum_j w_{ij}^{x^t} S_j^{t-1}),$$

$$\text{where } D_0(x) = \begin{cases} 0.8 & \text{if } x \geq 0.0 \\ 0.2 & \text{if } x < 0.0. \end{cases}$$

TABLE I
EXPERIMENTAL RESULTS FROM TRAINING THE DISCRETE RECURRENT NETWORK ON REGULAR GRAMMARS.*

Grammar	Training set # of strings	Training set L_{max}	# of hidden units	Mean # of epochs	σ of epochs	Mean # of total characters
Tomita #1	50	5	4	36.4	33.4	5205
Tomita #2	100	8	4	38.8	27.3	18120
Tomita #3	150	12	4	82.8	43.2	77040
Tomita #4	100	8	4	76.6	27.0	31712
Tomita #5	100	8	4	64.4	20.7	26662
Tomita #6	100	8	4	20.8	8.3	8611
Tomita #7	100	10	4	138.5	31.1	70774
Vending machine	365	6	5	231.8	22.4	383165
10-state machine	317	12	8	5798	—	14315262

*L_{max} is the maximum length of training strings. The numbers for epochs and total characters processed during learning are the average numbers over 5 runs with different random weight initializations, except for the 10-state machine, for which only one run was obtained. σ is the standard deviation of the epochs over the 5 runs. All runs, except one in learning Tomita #3 and one in learning Tomita #7 which failed to converge, have perfect generalization performance; i.e., are 100% correct on strings of any length.

(Here x^t is the input bit at time step t.)

Hence, the sigmoid units can be eliminated during testing to simplify computation.

During training, however, the gradient of the soft sigmoid function is made use of in a pseudo-gradient method for updating the weights.

B. The Pseudo-Gradient Learning Method

In order to train the discrete network, in [18] we proposed an approximation to gradient descent that we call the pseudo-gradient learning rule. During training, at the end of each string $\{x^0, x^1, \cdots, x^L\}$ the mean squared error is calculated as follows (note that L is the string length and that h_0^L is the analog indicator value at the end of the string):

$$E = \frac{1}{2}(h_0^L - T)^2,$$

where

$$T = target = \begin{cases} 1 & \text{if "legal"} \\ 0 & \text{if "illegal."} \end{cases}$$

Update w_{ij}^n, the weight from unit j to unit i in **net** n, at the end of each string presentation:

$$w_{ij}^n = w_{ij}^n - \alpha \frac{\widetilde{\partial E}}{\partial w_{ij}^n}, \ \forall n, i, j,$$

$$\frac{\widetilde{\partial E}}{\partial w_{ij}^n} = (h_0^L - T)\frac{\widetilde{\partial h_0}^L}{\partial w_{ij}^n}, \ \forall n, i, j,$$

where $\frac{\widetilde{\partial}}{\partial w_{ij}^n}$ is what we call the "pseudo-gradient" with respect to w_{ij}^n.

To get the pseudo-gradient $\frac{\widetilde{\partial h_0}^L}{\partial w_{ij}^n}$, pseudo-gradients $\frac{\widetilde{\partial h_k}^t}{\partial w_{ij}^n}$ for all t, k need to be calculated forward in time at each time step:

$$\frac{\widetilde{\partial h_k}^t}{\partial w_{ij}^n} = f' \cdot \Big(\sum_l w_{kl}^{x^t} \frac{\widetilde{\partial h_l}^{t-1}}{\partial w_{ij}^n} + \delta_{ki}\delta_{nx^t} S_j^{t-1}\Big), \forall k, t, w_{ij}^n. \quad (2)$$

(Initially, set: $\frac{\widetilde{\partial h_k}^0}{\partial w_{ij}^n} = 0, \forall i, j, n, k.$)

In carrying out the chain rule for the gradient we replace the real gradient $\frac{\partial S_l^{t-1}}{\partial w_{ij}^n}$, which is zero almost everywhere, by the pseudo-gradient $\frac{\widetilde{\partial h_l}^{t-1}}{\partial w_{ij}^n}$. The justification of the use of the pseudo-gradient is as follows: suppose we are standing on one side of the hard threshold function $S(x)$, at point $x_0 > 0$, and we wish to go downhill. The real gradient of $S(x)$ would not give any information since it is zero at x_0. If instead we look at the gradient of the function $f(x)$, which is positive at x_0 and increases as $x_0 \to 0$, it indicates that the downhill direction is to decrease x_0, which is also the case in $S(x)$. In addition, the magnitude of the gradient tells us how close we are to a step down in $S(x)$. Therefore, we can use that gradient as a heuristic hint to indicate direction and how close a step down would be. This heuristic hint is what we use as the pseudo-gradient in our gradient update calculation in (2).

C. Experimental Results on Learning Regular Grammars

Table I shows the experimental results obtained by training the discrete recurrent network by the pseudo-gradient learning method on various grammars, including the Tomita grammars [16], a simple vending machine model [18], and a particular 10-state machine [9] — some of the Tomita grammar and vending machine results are described in more detail in [18]. An epoch is one presentation of the whole training set to the network. The total number of characters processed is the cumulative count of all characters in all strings presented to the network in all training epochs. The results in Table I demonstrate that the discrete recurrent network model can be successfully trained to recognize simple grammars. Furthermore, because of its discrete nature, the network is inherently stable for strings of arbitrary length.

As an example of the ability of the network to learn grammars ofmedium complexity, Fig. 3(a) shows the effective automaton learned by the network trained on strings from the 10-state machine. Application of Moore's algorithm to this 15-state network automaton results in a reduction to the correct 10-state machine shown in Fig. 3(b).

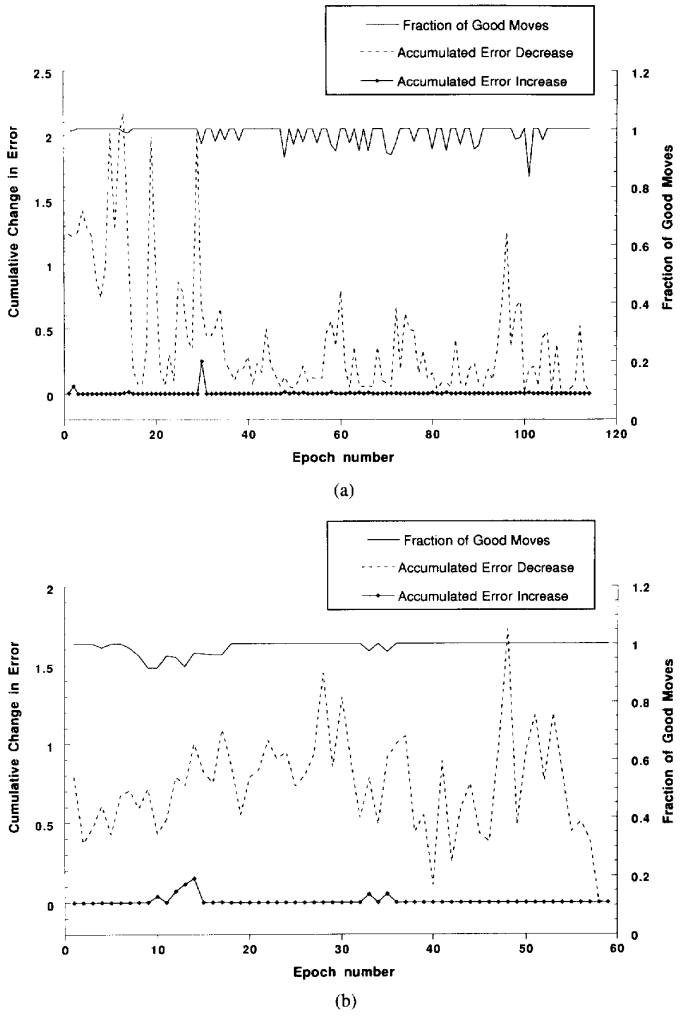


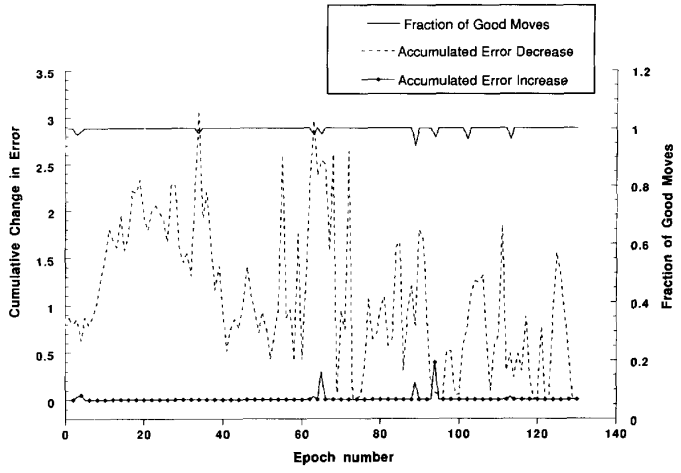
Fig. 4. Statistical record of the pseudo-gradient learning of regular grammars. In each plot, the solid curve corresponds to the fraction of successful moves for each learning epoch, the bottom dotted curve corresponds to the summation of error reduction on a string by all successful moves in each epoch, and the remaining curve is the summation of error increases on strings by all bad moves. The training set and the number of hidden units used for each grammar are the same as in Table I. The grammars being learned are: (a) Tomita #3. (b) Tomita #5. (c) Tomita #7. (d) The vending machine.

D. Empirical Investigation of the Pseudo-Gradient Learning

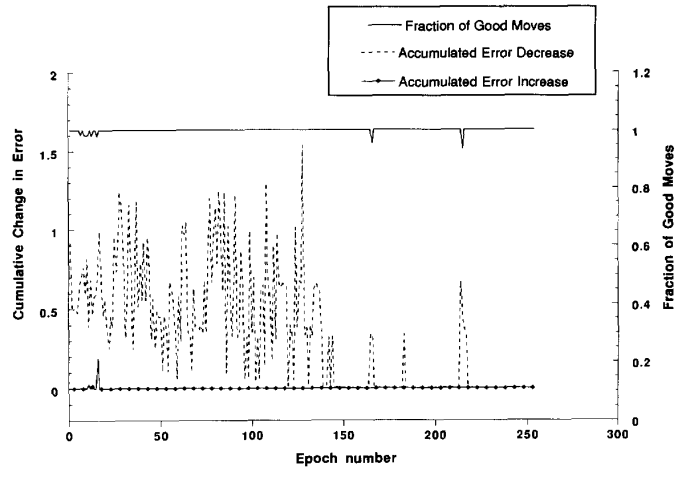
Theoretical analyses of learning in recurrent networks can be quitenon-trivial. In particular, analytical investigation of our proposed pseudo-gradient method for recurrent networks, appears intractable. Hence, we are limited to empirical evidence to support our claim that the method indeed appears to work well on non-trivial problems.

Figures 4(a)–(d) show the typical learning processes of four of the grammars described in Section II-C (plots of other grammars and of networks with different initial conditions have similar features and are not shown.) During each epoch of learning, the training strings are presented to the network one by one. In processing each string, pseudo-gradients are calculated and all weights are updated accordingly if the network makes an erroneous decision on that string. After each such weight update, we test the network with the new set of weights on the same string, and thus a new error is calculated. If the new error is smaller than the old one, we can then conclude that the pseudo-gradient has successfully decreased the error on this specific string as it was intended to; otherwise, we count it as a failure, or "bad move." Thus, the total fraction of "successful moves" (out of the total number of weight updates) induced by the pseudo-gradient algorithm can be calculated for each epoch. In each of the plots, the solid curve corresponds to this fraction of "good moves" as a function of the epoch number.

To evaluate the severity of the effect of all the "bad moves," we also record the magnitude of each error increase or decrease on a string after each weight update, and sum the error increases and decreases for each epoch. The lower dotted curve in each of the plots corresponds to the summation of all the error increases (or the cumulative effect of all "bad moves") as a function of epoch number, while the remaining



(c)



(d)

Fig. 4. *(Continued.)*

oscillating curve is the summation of all error decreases ("good moves") per epoch.

It is clearly evident that the pseudo-gradient algorithm induces successful moves over 80% of the time. In addition, when bad moves occur, their cumulative effect per epoch is always smaller (and often much smaller) than the cumulative effect of the successful moves, except during one epoch while learning the Tomita #7 grammar. Hence, the empirical evidence clearly indicates that the overwhelming tendency of the pseudo-gradient algorithm is to reduce the error on a per-string and per-epoch basis.

It is interesting to note that when one looks at the bad moves individually, the magnitude of an error increase by a single bad move is on average much larger than an error decrease caused by a single successful move—however, it is the *cumulative* effect that accounts for the convergence of the learning. Also note that the bad moves do not necessarily occur more frequently as the grammars become more complicated.

In conclusion, our empirical investigations have shown that although following the pseudo-gradient descent direction does not guarantee error reduction, it is certainly an effective way to conduct the training of discrete recurrent networks.

III. Discrete Recurrent Networks with External Stacks

Regular grammars are the simplest type of grammar in the Chomsky language hierarchy [10], and have a one-to-one correspondence with finite-state machines. Thus, a network that can represent any finite state machine is sufficient for representing regular grammars. The next class of grammars in the hierarchy are called context-free or type 2 grammars. They represent a much wider class of languages than do regular grammars—finite-state machines are not sufficient to represent all such grammars.

The theory of finite automata and formal languages states that there exists a one-to-one correspondence between context-free languages and pushdown automata. That is, one needs to have an external stack to operate on beside the finite-state machine in order to represent context-free grammars. By training the network to behave like a pushdown automaton

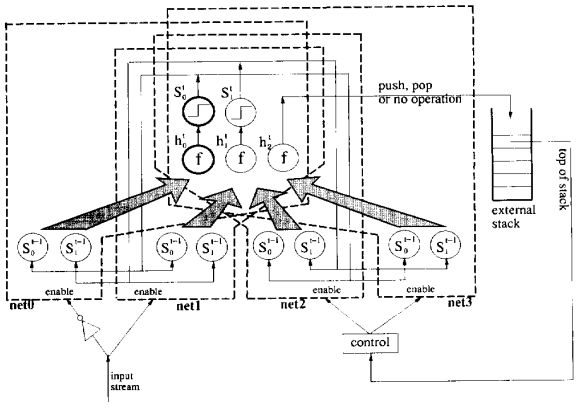


Fig. 5. A discretized second-order network with an external stack.The thick circled unit h_0^t is the indicator unit: $h_0^t > 0.5$ for legal strings and $h_0^t < 0.5$ for illegal strings.

we equivalently obtain a finite-state machine with an external stack that accepts the corresponding context-free grammar.

As in [3], we restrict the scope to context-free grammars with the following restrictions: given a current automaton state, there cannot be more than one choice of next state, the alphabet of the stack symbol is set to be the same as the input alphabet, only the current input symbol can be pushed onto the stack, and epsilon transitions (which can make state transitions or stack actions without reading in a new input symbol) are not allowed. In short, we consider a subset of deterministic pushdown automata, or deterministic context-free grammars. Note that to learn an arbitrary push-down automaton, a third-order architecture is necessary [3].

Shown in Fig. 5 is the structure of a discrete recurrent network withan external stack for the case of binary input and stack symbols. The primary differences between this structure and the one proposed in [3] are that we have a discrete stack as well as discretized units.

In Fig. 5 we have, in effect, four first-order networks with shared hidden units. In addition to the input symbol that acts as control to enable or disable **net0** or **net1**, the current top-of-stack symbol also acts as a second gating control that enables or disables **net2** or **net3**. Note that if the stack is empty, then both **net2** and **net3** are disabled, a situation that does not happen to the **net0–net1** pair.

As before, the unit h_0 is defined to be the "indicator" unit, whose activation should be greater than 0.5 at the end of a legal string and smaller than 0.5 otherwise. The last unit, in this case h_2, is singled out to be the "action" unit, whose activation decides what stack action to take. However, the value of this activation does not get copied back to the next time step. If h_2 is greater than a certain value (for the experiments reported here it is set to 0.6) then the current input symbol is pushed to the stack. If it is smaller than a certain value (0.4 in our case), then a symbol is popped out of the stack. Otherwise no action is taken.

The activation functions of the h units and the discretization function of the S units are the same as defined in Section II.

The error functions for training networks with stacks to learn context-free grammars are more complicated than for the simple grammars discussed in Section IV. Several situations can be encountered during learning, each requiring the use of a different error function. We start by basing our error functions on those proposed in [3], but there are some significant differences.

Let $h_0, h_1, ..., h_N$ be the hidden units of the network, where h_0 is the "indicator" unit and h_N is the "action" unit. Assume the current string being processed is $x^0, x^1, ..., x^L$, where L is the string length. Let d^t denote the depth of the stack at time step t, and let a^t be the top of stack symbol at time step t. The different error functions are as follows:

1) If the string is legal and the end of the string is reached (without any attempt to pop an empty stack),

$$E = \frac{1}{2}((1 - h_0^L)^2 + (d^L)^2).$$

This means that for legal strings we want both the indicator unit to be on and the stack to be empty.

2) If the string is illegal and the end of string is reached (without any attempt to pop an empty stack),

$$E = \begin{cases} h_0^L - d^L & \text{if } h_0^L - d^L > 0 \\ 0 & \text{otherwise.} \end{cases}$$

This means that for illegal strings we want either the stack to be nonempty, or the indicator unit to be off.

3) If the network attempts to pop an empty stack at time step t,

$$E = \begin{cases} \frac{1}{2}(1 - h_N^t)^2 - d^t & \text{if the string is legal} \\ 0 & \text{if the string is illegal.} \end{cases}$$

This means that for legal strings we want to correct the error of attempting to pop an empty stack by forcing the action unit value away from 0; i.e., avoid the "pop stack" action and at the same time encourage the stack to become nonempty. On the other hand, for illegal strings, we do nothing because the attempt to pop an empty stack is considered an indication that the string is illegal.

Das *et al.* have suggested in [3] that by providing the network with a "teacher" or an "oracle" to give hints, the learning can be sped up significantly. The teacher or oracle works as follows: there are certain illegal strings that are not prefixes to any legal strings; i.e., any symbols that follow such strings do not provide any further information. Henceforth, we will call these strings dead strings. The teacher is assumed to have the ability to identify such strings. Whenever a point is reached in the input string such that no further processing of the remaining string is necessary, the teacher produces a signal and the learning is halted. The network is then trained to have another special hidden unit, designated as the "dead unit," turn on. After the network has been trained in this way, a string is considered to be classified as illegal whenever the dead unit is turned on during testing. The error functions have to be modified accordingly.

We found that it is not sufficient to add an error function only for the dead strings and to keep the other error functions (3)–(5) the same. For strings other than the dead strings, the

network needs to be trained to have the dead unit turn off to avoid confusion. More specifically, letting h_1^t be the dead unit, we have the following:

1) If the string is legal and the end of string is reached (without any attempt to pop an empty stack),

$$E = \frac{1}{2}((1 - h_0^L)^2 + (d^L)^2 + (h_1^L)^2),$$

i.e., we want the indicator unit to be on, the stack to be empty *and the dead unit to be off.*

2) If the string is illegal but not a dead string, and the end of string is reached (without any attempt to pop an empty stack),

$$E = \begin{cases} h_0^L - d^L + \frac{1}{2}(h_1^L)^2 & \text{if } h_0^L - d^L > 0 \\ \frac{1}{2}(h_1^L)^2 & \text{otherwise,} \end{cases}$$

i.e., we want either the stack to be nonempty, or the indicator unit to be off, *and for both cases, the dead unit to be off.* The dead unit should not be on for such strings because they could be prefixes to certain legal strings.

3) If the string up to time step t is a dead string,

$$E = \begin{cases} \frac{1}{2}((1 - h_1^t)^2 + (h_0^t)^2) & \text{if stack is empty} \\ \frac{1}{2}(1 - h_1^t)^2 & \text{otherwise.} \end{cases}$$

This means we want the dead unit to turn on *and either the indicator unit to turn off or the stack to be nonempty.*

4) If the dead unit turns on at time step t before any possible signal for a dead string,

$$E = \frac{1}{2}(h_1^t)^2.$$

We do not want the dead unit to turn on too early since the string up to thus point could still be a prefix to certain legal strings.

5) If the network attempts to pop an empty stack at time step t, before any possible signal for a dead string,

$$E = \begin{cases} \frac{1}{2}(1 - h_N^t)^2 - d^t & \text{if the string is legal} \\ 0 & \text{if the string is illegal.} \end{cases}$$

Here we do not try to force the dead unit to turn on or off because it has been behaving as desired so far.

As in Section III, for the case with non-stack networks, the pseudo-gradient method is again used for training. The pseudo-gradients of error functions in weight space concern both $\frac{\widetilde{\partial h_k}^t}{\partial w_{ij}^n}$ for all t, k, n, i, j, and $\frac{\widetilde{\partial d}^t}{\partial w_{ij}^n}$ for all t, n, i, j. The former is calculated the same way as before. To calculate the latter, i.e., the pseudo-gradient of the depth of the stack, we use the iterative operational equation:

$$d^t = d^{t-1} + D_1(h_N^t),$$

where

$$D_1(x) = \begin{cases} 1 & \text{if } x > 0.4 \\ -1 & \text{if } x < 0.6 \\ 0 & \text{otherwise.} \end{cases}$$

Initially, set $\frac{\widetilde{\partial d}^0}{\partial w_{ij}^n} = 0$ for all n, i, j. After each time step, update:

$$\frac{\widetilde{\partial d}^t}{\partial w_{ij}^n} = \frac{\widetilde{\partial d}^{t-1}}{\partial w_{ij}^n} + \frac{\widetilde{\partial h_N}^t}{\partial w_{ij}^n}, \ \forall n, i, j.$$

Here, in place of the gradient of the piece-wise step function D_1, we still use the pseudo-gradient of the action unit h_N. Although the value of the action unit does not get discretized and copied back after each time step, its pseudo-gradient can still be calculated by utilizing the pseudo-gradients of other hidden units:

$$\frac{\widetilde{\partial h_N}^t}{\partial w_{ij}^n} = f' \cdot \Big(\sum_{l=0}^{N-1} w_{Nl}^{x^t} \frac{\widetilde{\partial h_l}^{t-1}}{\partial w_{ij}^n} + \sum_{l=0}^{N-1} w_{Nl}^{x^t} \frac{\widetilde{\partial h_l}^{t-1}}{\partial w_{ij}^n} + \delta_{Ni}\delta_{nx^t} S_j^{t-1}\Big), \ \forall t, w_{ij}^n.$$

Here we have left out a term concerning the top-of-stack symbol's dependency on the weights. Since a simple recurrent form of this term is analytically impossible to derive, an approximation was used in [15]. In our formula, the pseudo-gradient is itself an approximation; further fine tuning by this term may not be necessary. Empirical results in Section IV will demonstrate that the networks can indeed perform successful learning without this term in the formula. Thus, the coupling between the stack and the network during learning is reflected only in the previous formula for the gradient of the stack depth.

IV. Experimental Results on Learning Deterministic Context-Free Grammars

A. Overall Results

We experimented with the same grammars as in [3], i.e.,

1) The parenthesis matching grammar.
2) The postfix grammar.
3) $a^n b^n$.
4) $a^{m+n} b^m c^n$.
5) $a^n b^n c b^m a^m$.

As in [3], a training set consists of all strings up to a certain length, with repeated legal strings so that there are about half as many legal strings as illegal ones.

II(a) and (b) show the detailed results for experiments with and without hints, respectively. The numbers in each row are averages over the successful runs (out of 10 possible successful runs) with different initial conditions—a successful run is taken to mean a run in which the network generalizes perfectly for all string lengths. The number of overfitting runs indicates the number of times in the 10 runs that the network overfits the data by using too many internal states and did not generalize. The number of non-convergent runs is the number of times in the 10 runs that the training had not converged after 1000 epochs and was halted. Note that the number of unsuccessful runs are significantly fewer for the case with hints than without hints—hence, hints generally improve the reliability of the learning procedure. It is still an open question as to how to avoid overfitting in general by controlling the

Table II. Experimental results from training the discrete recurrent network on context-free grammars (a) with hints; (b) without hints.*

(a)

grammar	training set # of strings	training set L_{max}	# of hidden units	N_n	N_o	N_s	mean of epochs	σ of epochs	mean # of total characters
Parenthesis	46	6	3	0	0	10	28.8	16.3	5205
Postfix	63	7	4	1	0	9	62.3	17.1	21131
$a^n b^n$	32	6	4	2	0	8	127.3	4.9	16797
$a^{m+n} b^m c^n$	120	8	5	2	0	8	63	36.0	7560
$a^n b^n c b^m a^m$	150	7	7	3	3	4	328.8	249.1	243275

(b)

grammar	*training set* # of strings	*training set* L_{max}	# of hidden units	N_n	N_o	N_s	mean # of epochs	σ of epochs	mean # of total character
Parenthesis	180	6	3	0	0	10	12.0	10.5	11208
Postfix	371	7	4	4	2	4	185.8	149.0	408464
$a^n b^n$	760	8	5	4	4	2	150.5	87.5	793436

* The training set and hidden unit columns indicate the fixed learning parameters for each grammar. 10 runs with different random initial weights were carried out for each grammar. N_s, the number of successful runs is the number of runs (of the 10 possible) for which the trained network generalized perfectly for strings of any length. The means for the epochs and total characters processed (and the standard deviation for the epochs) were estimated only from the successful runs. N_o, the number of overfitting runs is the number where the network overfitted the data and did not generalize perfectly. N_n, the number of non-convergent runs is the number of runs where the network did not converge on the training data after 1000 epochs.

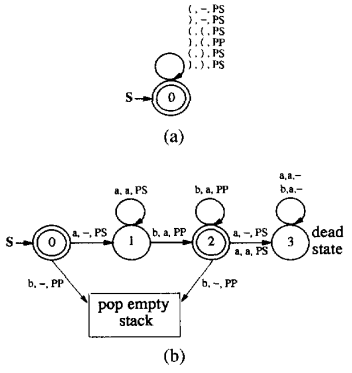


Fig. 6. Extracted pushdown automata from the discretized network withan external stack after learning (a) the parenthesis grammar without hints; (b) the grammar $a^n b^n$ with hints. Double-circled means the state has an indicator unit on, $S_0 = 0.8$; thus, a processed string is legal if the automaton arrives at such a state *and* if the stack is empty. A dead state means the state has its dead unit on, $S_1 = 0.8$; a processed string is illegal as soon as the automaton arrives at such a state. A transition rule is labeled by "x,y,z," where x stands for the current input symbol, y stands for the top-of-stack symbol ("-" means an empty stack), and z stands for the operation taken on the stack; "PS" means push, "PP" means pop.

size of the derived automaton during learning. It should be noted however that overfitting did not occur for 4 out of the 5 grammars in the experiments when hints were provided.

The hidden unit sizes and training set sizes shown in Table II(a) and (b) are the minimum sizes for which generalization could be obtained for each problem—experiments using either less training data or fewer hidden units invariably resulted in less than perfect generalization.

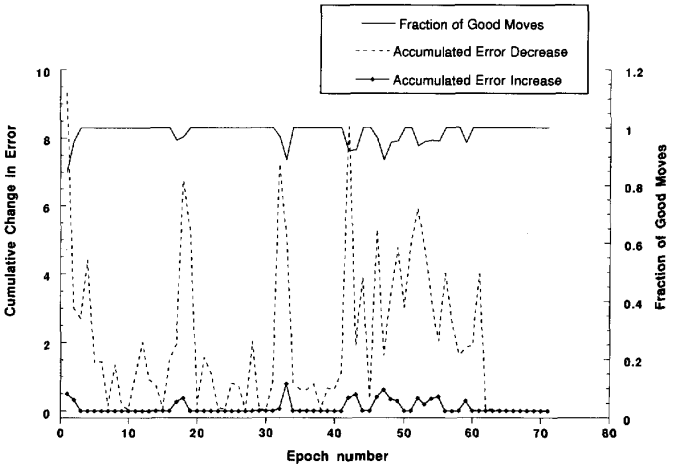
As an example, Fig. 6(a) and (b) show the derived pushdown automata from the networks after being trained on the parenthesis-matching grammar and the $a^n b^n$ grammar respectively. As before, each state corresponds to one single point in the network's hidden unit activation space and the transition rules are derived similarly: set the S_i^{t-1} units to each of the points (states) in the activation space, give the network different combinations of input and top-of-stack controls, and thus calculate the next state given such input and stack conditions.

Note that for the parenthesis-matching grammar, the network finds a pushdown automaton that has one single state. Starting from an empty stack, when the input is a "(," it pushes this input onto the stack. When the input is a "), " it either pops a "(" from the stack if the top-of-stack is a "(," or pushes the ")" onto the stack otherwise. Thus, whenever there are more ")"s than "("s, the machine executes a "push stack" operation no matter what the input symbol is, making the stack nonempty (indicating an illegal string) from this point on.

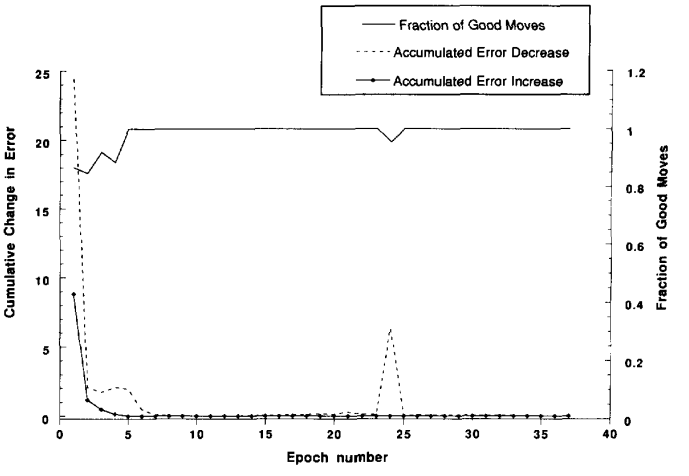
B. Empirical Investigation of the Pseudo-Gradient Learning

In a manner similar to that of Section II-D, we investigated how well the pseudo-gradient learning performed in learning pushdown automata. Plots of the fraction of successful moves by the pseudo-gradient algorithm and the accumulated error increases and decreases as a function of epoch number are shown in Fig. 7.

It can be observed from the plots that the pseudo-gradient algorithm makes bad moves in learning context-free grammars more often than it did in learning regular grammars. However, the percentage of successful moves are still mostly over 80%,



(a) Postfix, with hint.



(b) $a^{m+n}b^m c^n$, with hint.

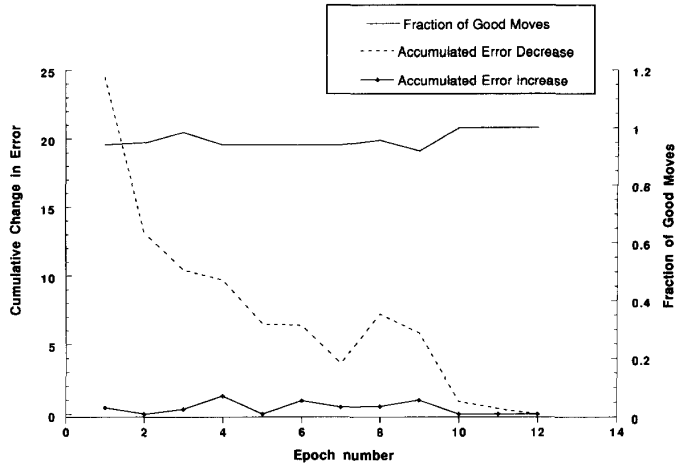
Fig. 7. Statistical record of the pseudo-gradient learning of pushdown automata. In each plot, the solid curve corresponds to the fraction of successful moves for each learning epoch. The bottom dotted curve corresponds to the summation of error reduction on a string by all successful moves in each epoch, and the remaining curve is the summation of error increases on strings by all bad moves. The training set and the number of hidden units used for each grammar are the same as in Table II. The grammars being learned are: (a) Postfix, with hint. (b) $a^{m+n}b^m c^n$, with hint. (c) Parenthesis matching, without hint. (d) $a^n b^n$, without hint.

and the accumulated error increases (due to bad moves) for any epoch are much smaller than the accumulated error decreases, except during one epoch in learning $a^n b^n$ without hints. Thus, as we found with the regular grammars, the empirical evidence suggests that the pseudo-gradient algorithm is quite effective in training discrete recurrent networks with external stacks.

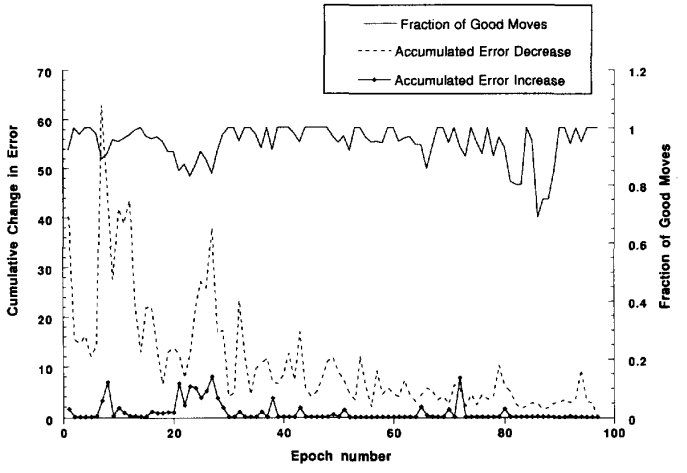
C. Discussion

Using a discrete network as well as a discrete stack results in the advantages of a stable network, and a clear understanding of the operation of the stack. In [3], where a continuous stack was used, the results show that the trained networks do not always generalize perfectly.

From the results in Table II, it can be seen that providing the network with hints can indeed speed up learning, or even enable the learning of the grammars in cases where the grammar could not be learned without hints. However, unlike [3], we did not find incremental presentation of the training data helped in improving the learning. Incremental presentation means that the network is initially given a small data set consisting of only short strings. After it has learned the current data set, more strings longer in length are added to the training set until all training strings are learned. We found in our experiments that once the network finds a configuration to fit the small data set with short strings, it is sometimes very hard to drag it away from that configuration to a desired configuration that will fit the later (longer) strings as well. The training times with and without incremental presentation



(c) Parenthesis matching, without hint.



(d) $a^n b^n$, without hint.

Fig. 7. *(Continued.)*

of strings are comparable in our experiments. The numbers listed in Table II(a) and (b) are of runs with the training data set presented to the network all at once.

We postulate that the reason why incremental learning worked for analog networks but not for discrete networks is due to the nature of analog and discrete networks. The analog network always finds a "soft" solution to a data set, which only has clear decisions for short strings, but is vague on long strings. Thus it is easy for it to "harden" such a solution when more restrictions about longer strings are enforced. The result is a solution whose "hardness" or decisiveness depends on the maximum length of the training strings. On the other hand, the discrete network always finds a "hard" solution to a data set that has clear decisions for strings of any length. Once it settles in such a solution it is hard to enforce restrictions about longer strings that contradict the current solution. So one may as well provide all the restrictions to the network at once. As long as there exists sufficient information in the data set, the resulting solution does not depend on the maximum length of training strings.

V. Conclusion

The primary advantages of introducing discretization into recurrentnetworks can be summarized as follows:

1) Once the network has successfully learned an automaton from the training set, its internal states are stable. The network will always classify input strings correctly, independent of the lengths of these strings.
2) No manual clustering (as in [8]) is required to extract the state machine explicitly, since instead of using "cluster clouds" as its state representation, the network forms distinct, isolated points as states. Each point in activation space is a *distinct* state and, hence, the trained network behaves *exactly* like a state machine.
3) In terms of implementation the discretized recurrent network is easier to implement in hardware particularly when an external stack is used.

In conclusion, we have presented in this paper the basic ideas and algorithms for implementing stable discrete recurrent networks for learning deterministic context-free grammars.

Specifically, we extended our previous discrete network models to include an external discrete stack with discrete symbols, defined an appropriate error function for learning, and derived a pseudo-gradient learning rule for this error function. The available empirical evidence indicates that the pseudo-gradient learning algorithm is effective in training such a network. The overall experimental results show that the proposed network has similar capabilities for learning context-free grammars as the *analog* second-order networks, while avoiding any problems with instability on long strings.

REFERENCES

[1] J. Carroll, D. Long, *Theory of Finite Automata*, Englewood Cliffs, NJ: Prentice Hall, 1989.

[2] A. Cleeremans, D. Servan-Schreiber, and J. L. McClelland, "Finite state automata and simple recurrent networks," *Neural Computation*, vol. 1, pp. 372–381, 1989.

[3] S. Das, C. L. Giles, and G. Z. Sun, "Using prior knowledge in an NNPDA to learn context-free languages," in *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan and C. L. Giles, Eds. San Mateo, CA: Morgan Kaufmann, pp. 65–72, 1993.

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

[5] J. L. Elman, "Distributed representations, simple recurrent networks, and grammatical structure," *Machine Learning*, vol. 7, nos. 2 and 3, pp. 195–225, 1991.

[6] S. E. Fahlman, "The recurrent cascade-correlation architecture," *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody, and D. S. Touretzky, Eds. San Mateo, CA: Morgan Kaufmann, pp. 190–196, 1991.

[7] K. S. Fu, *Syntactic Pattern Recognition and Applications*, Englewood Cliffs, NJ: Prentice-Hall, 1982.

[8] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Computation*, vol. 4, no. 3, pp. 393–405, 1992.

[9] C. L. Giles, C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen, and Y. C. Lee, "Extracting and learning an unknown grammar with recurrent neural networks," in *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds. San Mateo, CA: Morgan Kaufmann, pp. 317–324, 1992.

[10] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Reading, MA: Addison-Wesley, 1979.

[11] M. I. Jordan, "Serial order: A parallel distributed processing approach," Tech. Rep. no. 8604, San Diego: University of California, Institute for Cognitive Science, 1986.

[12] J. B. Pollack, "The induction of dynamical recognizers," *Machine Learning*, vol. 7, nos. 2 and 3, pp. 227–252, 1991.

[13] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing*, pp. 354–361, Cambridge, MA: The MIT Press, 1986.

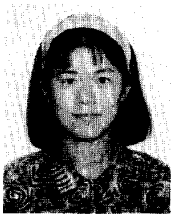
[14] D. Servan-Schreiber, A. Cleeremans, and J. L. McClelland, "Graded state machines: The representation of temporal contingencies in simple recurrent networks," *Machine Learning*, vol. 7, nos. 2 and 3, pp. 161–193, 1991.

[15] G. Z. Sun, H. H. Chen, C. L. Giles, Y. C. Lee, and D. Chen, "Connectionist pushdown automata that learn context-free grammars," *Proceedings of the International Joint Conference on Neural Networks*, vol. 1, pp. 577, Washington, DC, 1990.

[16] M. Tomita, "Dynamic construction of finite-state automata from examples using hill-climbing," *Proceedings of the Fourth Annual Cognitive Science Conference*, p. 105, 1982.

[17] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

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



Zheng Zeng was born in Nanjing, the People's Republic of China, in 1965. She received the B.S. degree in Electronics Engineering with honors from Tsinghua University, Beijing, the People's Republic of China in 1988, and the M.S. degree in Electrical Engineering from the California Institute of Technology, Pasadena, CA, in 1991, where she is currently pursuing the Ph.D. degree. Her research interests include machine learning techniques, neural network models and algorithms, their applications in pattern recognition, grammatical inference, and sequence analysis.

Rodney M. Goodman, (M'85), was born in London England, on February 22, 1947. He received the B.Sc. degree in Electrical Engineering from Leeds University, Yorkshire, U.K. in 1968, and the Ph.D. in Electronics at the University of Kent at Canterbury, U.K., in 1975. In 1985 Dr. Goodman joined the faculty of the Department of Electrical Engineering at the California Institute of Technology as Associate Professor. Dr Goodman's research interests are in error control coding, cryptography, neural networks, and expert systems—from both a theoretical and a VLSI implementation viewpoint. He has consulted for a wide variety of government and commercial organizations, and is a founder of two advanced technology research and development companies in the U.K. He is currently a consultant for the Jet Propulsion Laboratory, and Pacific Bell. Dr. Goodman is a Chartered Electrical Engineer of the I.E.E. in the U.K., and a Member of the IEEE.

Padhraic Smyth, (S'82–M'85–S'85–M'88), was born in Kilmovee, Ireland in 1962. He received a first-class honours B.E. (Bachelor of Engineering) degree from University College Galway, National University of Ireland, in 1984, and the M.S. and Ph.D. degrees in electrical engineering, in 1985 and 1988 respectively, from the California Institute of Technology. From 1985 to 1988 he worked part-time as a research consultant with Pacific Bell in the areas of telecommunications switching systems and automated network management. In 1988 he joined the Communications Systems Research Section at the Jet Propulsion Laboratory, Pasadena, where he is currently a Technical Group Leader. He has received 10 NASA certificates of appreciation for technical innovation since 1988 and was awarded the Lew Allen Prize for Excellence at JPL in 1993. Dr. Smyth's research interests include statistical pattern recognition, decision theory, information theory, source coding, telecommunications, and the application of probability and statistics to problems in artificial intelligence.