ENCODING SEQUENTIAL STRUCTURE IN SIMPLE RECURRENT NETWORKS

David Servan-Schreiber Axel Cleeremans James L. McClelland

Carnegie Mellon University

November 1988
CMU-CS-88-183

Abstract

We explore a network architecture introduced by Elman (1988) for predicting successive elements of a sequence. The network uses the pattern of activation over a set of hidden units from time-step t-1, together with element t, to predict element t+1. When the network is trained with strings from a particular finite-state grammar, it can learn to be a perfect finite-state recognizer for the grammar. When the net has a minimal number of hidden units, patterns on the hidden units come to correspond to the nodes of the grammar; however, this correspondence is not necessary for the network to act as a perfect finite-state recognizer. We explore the conditions under which the network can carry information about distant sequential contingencies across intervening elements to distant elements. Such information is maintained with relative ease if it is relevant at each intermediate step; it tends to be lost when intervening elements do not depend on it. At first glance this may suggest that such networks are not relevant to natural language, in which dependencies may span indefinite distances. However, embeddings in natural language are not completely independent of earlier information. The final simulation shows that long distance sequential contingencies can be encoded by the network even if only subtle statistical properties of embedded strings depend on the early information.

David Servan-Schreiber was supported by an NIMH Individual Fellow Award MH-O9696-01. Axel Cleeremans was supported in part by a fellowship from the Belgian American Educational Foundation, and in part by a grant from the National Fund for Scientific Research (Belgium). James L. McClelland was supported by an NIMH Research Scientist Career Development Award MH-00385. Support for computational resources was provided by NSF(BNS-86-09729) and ONR(N00014-86-G-0146)

Contents

1. Introduction 3

2. Learning a finite-state grammar 5
2.1 Material and task 5
2.2 Network architecture 5
2.3 Coding of the strings 6
2.4 Training 6
2.5 Performance 7
2.6 Analysis of internal representations 11

3. Discovering and using path information 15
3.1 Material 15
3.2 Network architecture 16
3.3 Performance 16
3.4 Is the simple recurrent network still a finite-state machine ? 19

4. Learning 21
4.1 Discovering letters 22
4.2 Discovering arcs 26
4.3 Encoding the path 26

5. Encoding non-local context 28
5.1 Processing loops 28
5.2 Spanning embedded sequences 30

6. Discussion 34

7. References 37

Encoding Sequential Structure in Simple Recurrent Networks

1. Introduction

As language abundantly illustrates, the meaning of individual events in a stream --such as words in a sentence-- is often determined by preceding events in the sequence which provide a context. The word 'ball' is interpreted differently in "The countess threw the ball" and in "The pitcher threw the ball". Similarly, goal-directed behaviors and planning are characterized by coordination of behaviors over long sequences of input-output pairings, again implying that goals and plans act as a context for the interpretation and generation of individual events.

The similarity-based style of processing of connectionist models provides natural primitives to implement the role of context in the selection of meaning and actions. However, most connectionist models of sequence processing present all cues of a sequence in parallel and often assume a fixed length for the sequence (e.g, Cottrell, 1985; Fanty, 1985; Selman, 1985; Sejnowski & Rosenberg, 1986; Hanson and Kegl, 1987). Typically, these models use a pool of input units for the event present at time **t**, another pool for event **t+1**, and so on, in what is often called a 'moving window' paradigm. As Elman (1988) points out, such implementations are not psychologically satisfying, and they are also computationally wasteful since some unused pools of units must be kept available for the rare occasions when the longest sequences are presented.

Some connectionist architectures have specifically addressed the problem of learning and representing the information contained in sequences in more elegant ways. Jordan (1986) described a network in which the output associated to each state was fed back and blended with the input representing the next state (Figure 1).

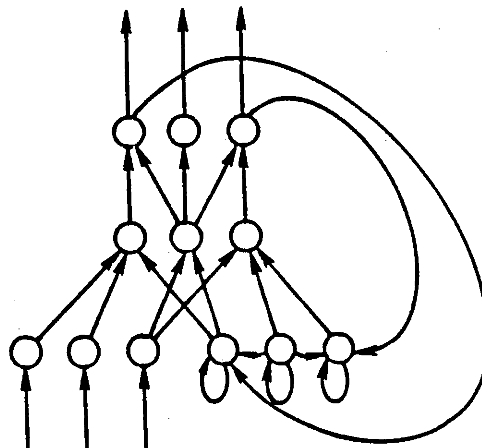

Figure 1
The Jordan (1986) Sequential Network

After several steps of processing, the pattern present on the input units is characteristic of the particular sequence of states that the network has traversed. With sequences of increasing length, the network has more difficulty discriminating on the basis of the first cues presented, but the architecture does not rigidly constrain the length of input sequences. However, while such a network learns *how to use* the representation of successive states, it does not discover a representation for the sequence.

Elman (1988) has studied a slightly different architecture. In this simple recurrent network (Figure 2), the hidden unit layer is allowed to feed back on itself, so that the intermediate results of processing at time **t-1** can influence the intermediate results of processing at time **t**. In practice, the simple recurrent network is implemented by copying the pattern of activation on the hidden units onto a set of 'context units' which feed into the hidden layer along with the input units.

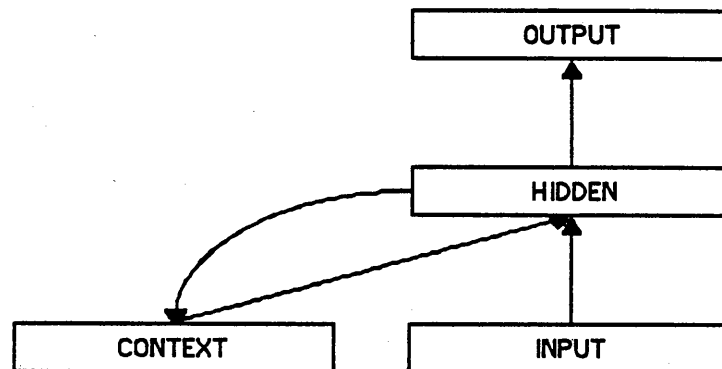

Figure 2
The Simple Recurrent Network.
Each box represents a pool of units and each forward arrow represents a complete set of trainable connections from each sending unit to each receiving unit in the next pool. The backward arrow from the hidden layer to the context layer denotes a copy operation.

In Elman's simple recurrent networks the set of context units provides the system with memory in the form of a trace of processing at the previous time slice. As Rumelhart, Hinton and Williams (1986) have pointed out, the pattern of activation on the hidden units corresponds to an 'encoding' or 'internal representation' of the input pattern. By the nature of back-propagation, such representations correspond to the input pattern partially processed into features relevant to the task (e.g, the 'Italian' and 'English' features in Hinton's family tree example, 1986). In the recurrent networks, internal representations encode not only the prior event but also relevant aspects of the representation that was constructed in predicting the prior event from its predecessor. When fed back as input, these representations could provide information that allows the network to maintain prediction-relevant features of an entire sequence.

The purpose of the present study is to describe and understand what type of representations develop in the simple recurrent network, and the steps through which they are acquired. In the following (section 2), we start by examining the performance of a simple recurrent network on a very large set of sequences generated at random by a finite-state grammar.

We suggest that the the network comes to approximate the finite-state automaton corresponding to the grammar. In section 3, we examine the network's performance on a more constrained training set obtained by limiting the length of the sequences generated by the same finite-state grammar. Section 4 is dedicated to learning. We examine in detail how the network develops internal representations that encode relevant information about the entire sequence of elements it has been presented with. Finally, we discuss more specifically the network's performance on complex sequences involving embedded clauses, and assess the relevance of this research with regard to language processing (sections 5 and 6). We also discuss conditions under which the learning task becomes harder or impossible.

2. Learning a finite state grammar

2.1 Material and task

In all of the following explorations, the network is assigned the task of predicting successive elements of a sequence. This task is interesting because it allows us to examine precisely how the network extracts information about whole sequences without actually seeing more than two elements at a time. In addition, it is possible to manipulate precisely the nature of these sequences by constructing different training and testing sets of strings that require integration of more or less temporal information. The stimulus set thus needs to exhibit various interesting features with regard to the potentialities of the architecture (i.e. the sequences must be of different lengths, their elements should be more or less predictable in different contexts, loops and subloops should be allowed, etc.).

Reber (1967) used a small finite-state grammar in an artificial grammar learning experiment that is well suited to our purposes (Figure 3). Finite-state grammars consist of nodes connected by labeled arcs. A grammatical string is generated by entering the network through the 'start' node and by moving from node to node until the 'end' node is reached. Each transition from one node to another produces the letter corresponding to the label of the arc linking these two nodes. Examples of strings that can be generated by the above grammar are : TXS, PTVV, TSXXTVPS.

The difficulty in mastering the prediction task when letters of a string are presented individually is that two instances of the same letter may lead to different nodes and therefore different predictions about its successors. In order to perform the task adequately, it is thus necessary for the network to encode more than just the identity of the preceding letter.

2.2 Network Architecture

As illustrated in Figure 4, the network has a three-layer architecture. The input layer consists of two pools of units. The first pool is called the context pool, and its units are used to represent the temporal context by holding a copy of the hidden units' activation level at the previous time slice (Note that this is strictly equivalent to a fully connected feedback loop on the hidden layer). The second pool of input units represents the current element of the string. On each trial, the network is presented with an element of the string, and is supposed to produce the next element on the output layer. In both the input and the

output layers, letters are represented by the activation of a single unit. Five units therefore code for the five different possible letters in each of these two layers. In addition, two units code for *start* and *end* bits. These two bits are needed so that the network can be trained to predict the first element and the end of a string (Although only one *transition* bit is strictly necessary).
In this first experiment, the number of hidden units was set to 3. Other values will be reported as appropriate.

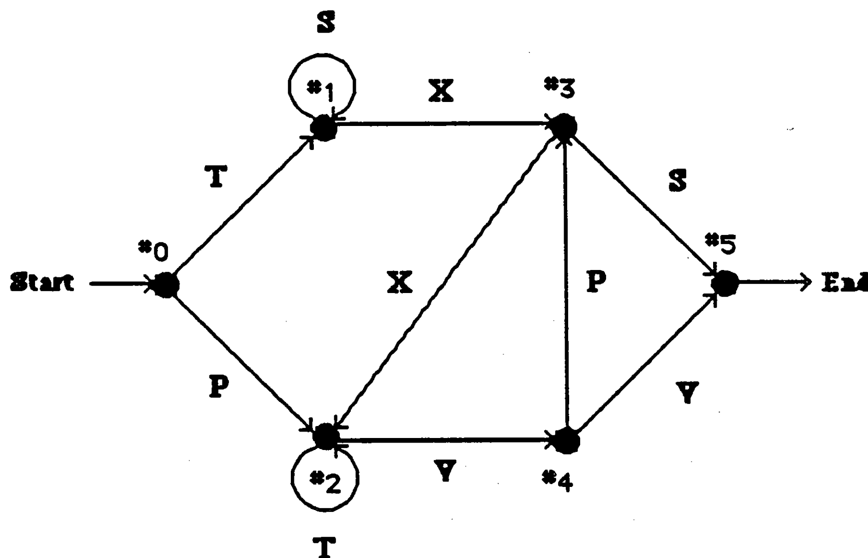

Figure 3
The finite-state grammar used by Reber (1967)

2.3 Coding of the strings

A string of **n** letters is coded as a series of **n+1** training patterns. Each pattern consists of two input vectors and one target vector. The two input vectors are :
1) A three-bit vector representing the activation of the hidden units at time **t-1**, and
2) A seven-bit vector representing element **t** of the string.
The target vector is a seven-bit vector representing element **t+1** of the string.

As the network should not rely on the context of the last element of a previous string when presented with the start bit of a new one, the activation of the context units is set to 0 in the very first pattern of each string.

2.4 Training

Training consisted of the presentation of 200,000 strings generated according to the finite-state grammar with no length constraints. The transition probabilities of all arcs were equal and set at 0.5. The average string was six letters long with a standard deviation of 6.9. All the weights in the network were initially set to random values between -0.5 and +0.5 After

the presentation of each letter in each string, an error signal was derived from comparing activation on the output layer (the network's prediction) with the target pattern determined by the next letter in the string. That error signal was back-propagated through all the forward-going connections and the weights adjusted before presenting the next letter.[1]

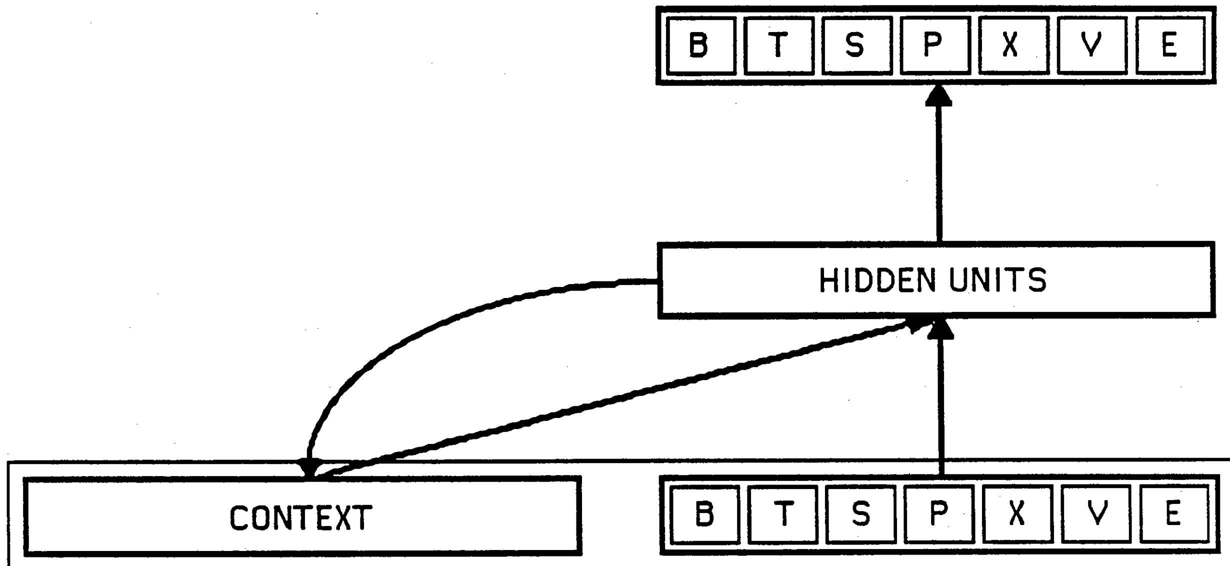

Figure 4
General architecture of the network

2.5 Performance

Figure 5 shows the state of activation of all the units in the network, after training, when the start symbol is presented (here the letter 'B' -- for begin). Activation of the output units indicate that the network is predicting two possible successors, the letters 'P' and 'T'. Note that the best possible prediction always activates two letters on the output layer except when the end of the string is predicted. Since during training 'P' and 'T' followed the start symbol equally often, each is activated partially in order to minimize error. Figure 6 shows the state of the network at the next time step in the string 'BTXXVV'. The pattern of activation on the context units is now a copy of the pattern generated previously on the hidden layer. The two successors predicted are 'X' and 'S'.

---

[1] Slightly modified versions of the BP program from McClelland and Rumelhart (1988) were used for this and all subsequent simulations reported in this paper. Values of learning rate and momentum (*eta* and *alpha* in Rumelhart et al. (1986)) were chosen sufficiently small to avoid large oscillations and were generally in the range of 0.01 to 0.02 for learning rate and 0.5 to 0.9 for momentum.

String Btxxvv

	B	T	S	P	X	V	E
output	0	53	0	38	1	2	0
hidden			1	0	10		
context			0	0	0		
	B	T	S	P	X	V	E
input	100	0	0	0	0	0	0

Figure 5
State of the network after presentation of the 'Begin' symbol.
Activation values are internally in the range 0 to 1.0 and are displayed on a scale from 0 to 100.
The capitalized letter indicates which letter is currently being presented on the input layer.

String bTxxvv

	B	T	S	P	X	V	E
output	0	1	39	0	56	0	0
hidden			84	0	28		
context			1	0	10		
	B	T	S	P	X	V	E
input	0	100	0	0	0	0	0

Figure 6
State of the network after presentation of an initial 'T'.
Note that the activation pattern on the context layer is identical to the activation pattern on the hidden layer at the previous time step.

The next two figures illustrate how the network is able to generate different predictions when presented with two instances of the same letter on the input layer in different contexts. In Figure 7, when the letter 'X' immediately follows 'T', the network predicts again 'S' and 'X' appropriately. However, as Figure 8 shows, when a second 'X' follows, the prediction changes radically as the network now expects 'T' or 'V'.

String	btXxvv						
	B	T	S	P	X	V	E
output	0	4	44	0	37	7	0
hidden			74	0	93		
context			84	0	28		
	B	T	S	P	X	V	E
input	0	0	0	0	100	0	0

Figure 7
State of the network after presentation of the first 'X'

String	btxXvv						
	B	T	S	P	X	V	E
output	0	50	1	1	0	55	0
hidden			6	9	99		
context			74	0	93		
	B	T	S	P	X	V	E
input	0	0	0	0	100	0	0

Figure 8
State of the network after presentation of the second 'X'

In order to test whether the network would generate similarly good predictions after every letter of any grammatical string, we tested its behavior on 20,000 strings derived randomly from the grammar. A prediction was considered accurate if, for every letter in a given string, activation of its successor was above 0.3. If this criterion was not met, presentation of the string was stopped and the string was considered 'rejected'. With this criterion, the network correctly 'accepted' all of the 20,000 strings presented.

Conversely, we presented the network with 130,000 strings generated from the same pool of letters but in a random manner -- i.e, mostly 'non-grammatical'. 0.2% (260) of these strings were in fact grammatical and the network accepted all of them. The remaining 99.8% of the strings were all ungrammatical and were all correctly rejected.

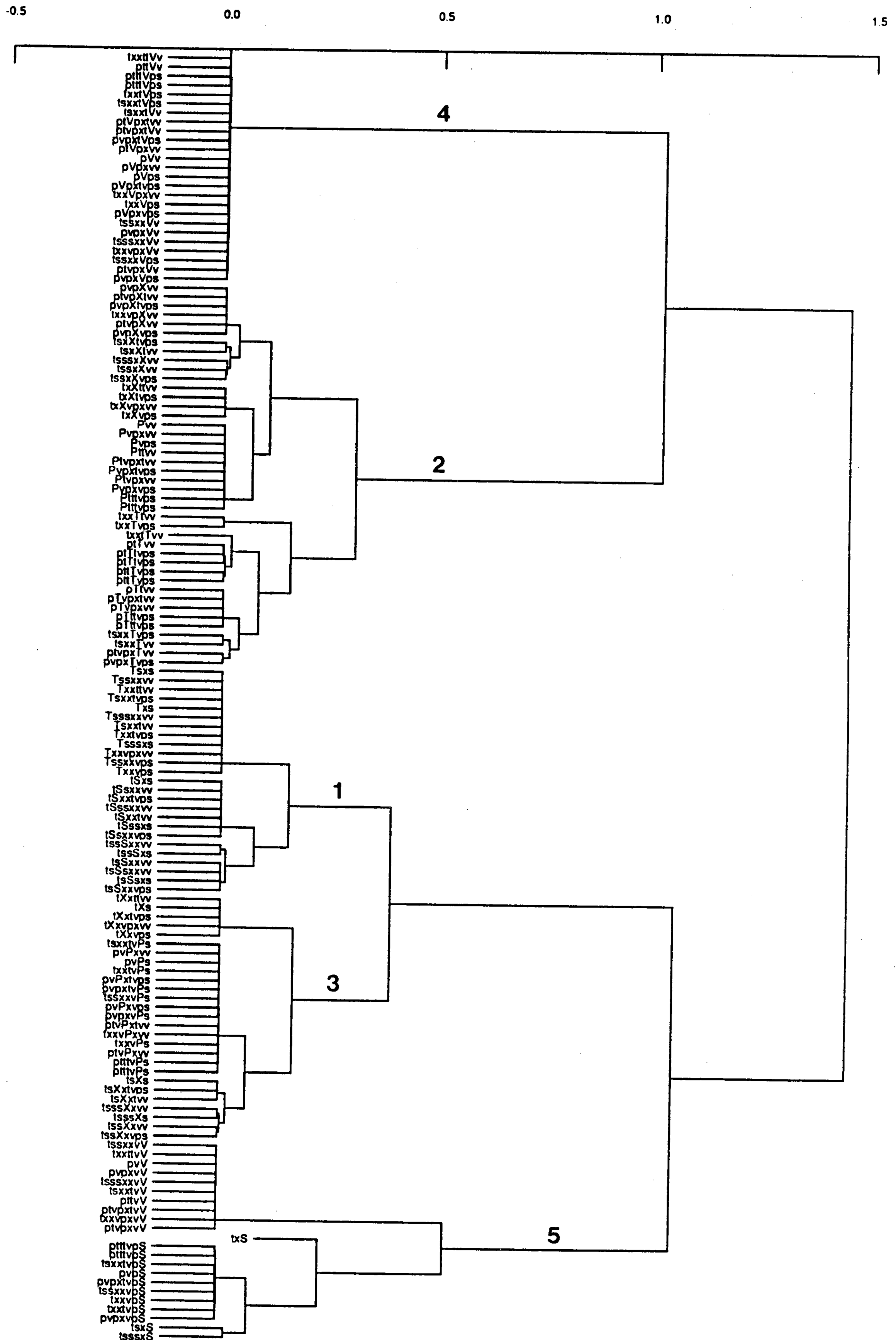

Figure 9
Hierarchical Cluster Analysis of the H.U. activation patterns after 200,000 presentations from strings generated at random according to the Reber grammar. (Three hidden units)

Finally, we presented the network with several extremely long strings such as:

'BTSSSSSSSSSSSSSSSSSSSSSSXXVPXVPXVPXVPXVPXVPXVPXVPXVPX
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTVPXVPXVPXVPXVPXVPS'

and observed that, at every step, the network correctly predicted both legal successors and no others.

Note that it is possible for a network with only three hidden units to reach this performance criterion with much less training. For example, one of our simulations reached criterion after 60,000 string presentations. In addition, increasing the number of hidden units also accelerates learning in terms of string presentations. A network with 15 hidden units reached criterion after 20,000 strings were presented. However, activation values on the output layer are not as clearly contrasted when training is less extensive.

2.6 Analysis of internal representations

Obviously, in order to perform accurately, the network takes advantage of the representations that have developed on the hidden units which are copied back onto the context layer. At any point in the sequence, these patterns must somehow encode the position of the current input in the grammar on which the network was trained. One approach to understanding how the network uses these patterns of activation is to perform a cluster analysis. We recorded the patterns of activation on the hidden units following the presentation of each letter in a small random set of grammatical strings. The matrix of Euclidean distances between each pair of vectors of activation served as input to a cluster analysis program[2]. The graphical result of this analysis is presented in Figure 9. Each leaf in the tree corresponds to a particular string, and the capitalized letter in that string indicates which letter has just been presented. For example, if the leaf is identified as 'pvPs', 'P' is the current letter and its predecessors were 'P' and 'V' (the correct prediction would thus be 'X or S').

From the figure, it is clear that activation patterns are grouped according to the different nodes in the finite state grammar: all the patterns that produce a similar prediction are grouped together, independently of the current letter. For example, the bottom cluster groups patterns that result in the activation of the 'End' unit (i.e, following the last 'V' or last 'S' in a string). Therefore, when one of the hidden layer patterns is copied back onto the context layer, the network is provided with information about the *current node*. That information is combined with input representing the *current letter* to produce a pattern on the hidden layer that is a representation of the *next node*. To a degree of approximation, the recurrent network behaves exactly like the finite state automaton defined by the grammar. It does does not use a stack or registers to provide contextual information but relies instead on simple state transitions, just like a finite state machine. Indeed, the network's perfect performance on randomly generated grammatical and non-grammatical strings shows that it can be used as a finite state recognizer.

---

[2] Cluster analysis is a method that finds the optimal partition of a set of vectors according to some measure of similarity (here, the euclidean distance). On the graphical representation of the obtained clusters, the contrast between two groups is indicated by the length of the vertical links. The length of horizontal links is not meaningful.

However, a closer look at the cluster analysis reveals that within a cluster corresponding to a particular node, patterns are further divided according to the path traversed before that node. For example, looking again at the bottom cluster, patterns produced by a 'VV', 'PS' and 'SXS' ending of the string are grouped separately by the analysis: they are more similar to each other than to the abstract prototypical pattern that would characterize the corresponding 'node'.[3] We can illustrate the behavior of the network with a specific example. When the first letter of the string 'BTX' is presented, the initial pattern on context units corresponds to node 0. This pattern together with the letter 'T' generates a hidden layer pattern corresponding to node 1. When that pattern is copied onto the context layer and the letter 'X' is presented, a new pattern corresponding to node 3 is produced on the hidden layer, and this pattern is in turn copied on the context units. If the network behaved *exactly* like a finite state automaton, the exact same patterns would be used during processing of the other strings 'BTSX' and 'BTSSX'. That behavior would be adequately captured by the transition network shown in Figure 10. However, since the cluster analysis shows that slightly different patterns are produced by the substrings 'BT', 'BTS' and 'BTSS', Figure 11 is a more accurate description of the network's state transitions. As states 1, 1' and 1" on the one hand and 3, 3' and 3" on the other are nevertheless very similar to each other, the finite state machine that the network implements can be said to approximate the idealization of a finite state automaton corresponding exactly to the grammar underlying the exemplars on which it has been trained.

However, we should point out that the close correspondence between representations and function obtained for the recurrent network with three hidden units is rather the exception than the rule. With only three hidden units, representational resources are so scarce that back-propagation forces the network to develop representations that yield a prediction on the basis of the current node alone, ignoring contributions from the path. This situation precludes the development of different -- redundant -- representations for a particular node that typically occurs with larger numbers of hidden units. When redundant representations do develop, the network's behavior still converges to the theoretical finite state automaton -- in the sense that it can still be used as a perfect finite state recognizer for strings generated from the corresponding grammar -- but internal representations do not correspond to that idealization. Figure 12 shows the cluster analysis obtained from a network with 15 hidden units after training on the same task. Only nodes 4 and 5 of the grammar seem to be represented by a unique 'prototype' on the hidden layer. Clusters corresponding to nodes 1, 2 and 3 are divided according to the preceding arc. Information about arcs is not relevant to the prediction task and the different clusters corresponding to the a single node play a redundant role.

Finally, preventing the development of redundant representations may also produce adverse effects. For example, in the Reber grammar, predictions following nodes 1 and 3 are identical ('X or S'). With some random sets of weights and training sequences, networks with only three hidden units occasionally develop almost identical representations for nodes 1 and 3, and are therefore unable to differentiate the first from the second 'X' in a string.

[3] This fact may seem surprising at first, since the learning algorithm does not apply pressure on the weights to generate different representations for different paths to the same node. Preserving that kind of information about the path does not contribute in itself to reducing error in the prediction task. We must therefore conclude that this differentiation is a direct consequence of the recurrent nature of the architecture rather than a consequence of back-propagation. We will indeed show that some amount of information about the path is encoded in the hidden layer patterns when a succession of letters is presented, even in the absence of any training. This important point will be examined in more detail in section 5.

In the next section we examine a different type of training environment, one in which information about the path traversed becomes relevant to the prediction task.

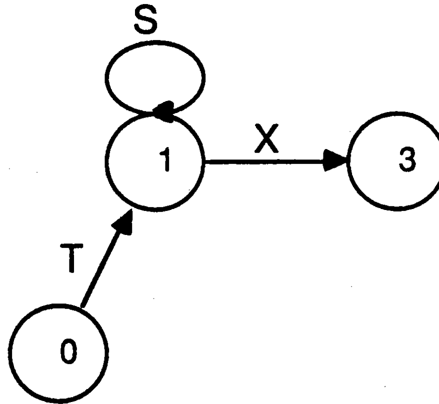

Figure 10
A transition network corresponding to the upper-left part of Reber's finite-state grammar

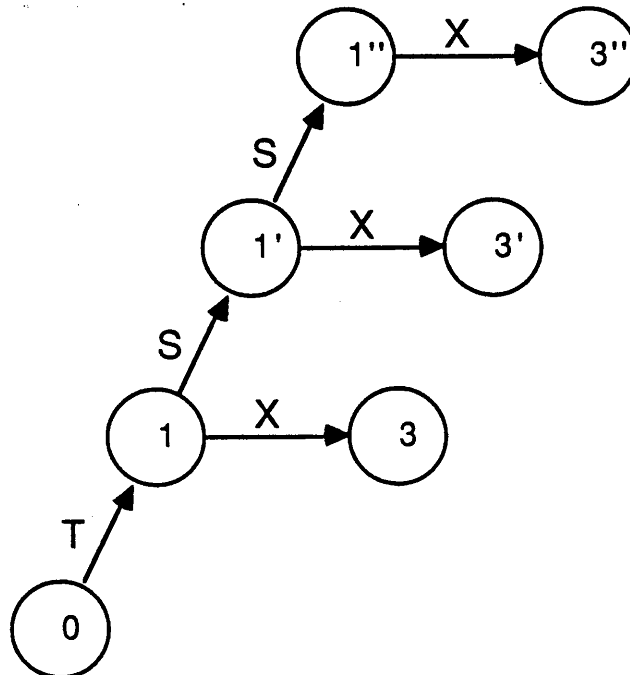

Figure 11
A transition network illustrating the network's true behavior

Figure 12
Hierarchical Cluster Analysis of the H.U. activation patterns after 200,000 presentations from strings generated at random according to the Reber grammar.
(Fifteen hidden units)

3. Discovering and using path information

The previous section has shown that simple recurrent networks can learn to encode the nodes of the grammar used to generate strings in the training set. However, this training material does not require information about arcs or sequences of arcs -- the 'path' -- to be maintained. How does the network's performance adjust as the training material involves more complex and subtle temporal contingencies ? We examine this question in the following section, using a training set that places many additional constraints on the prediction task.

3.1 Material

The set of strings that can be generated from the grammar is finite for a given length. For lengths 3 to 8, this amounts to 43 grammatical strings. The 21 strings shown in Figure 13 were selected and served as training set. The remaining 22 strings can be used to test generalization. The selected set of strings has a number of interesting properties with regard to exploring the network's performance on subtle temporal contingencies :

• As in the previous task, identical letters occur at different points in each string, and lead to different predictions about the identity of the successor. No stable prediction is therefore associated with any particular letter, and it is thus necessary to encode the position, or the node of the grammar.

• In this limited training set, length places additional constraints on the encoding because the possible predictions associated with a particular node in the grammar are dependent on the length of the sequence. The set of possible letters that follow a particular node depends on how many letters have already been presented. This information must therefore also be somehow represented during processing.

TSXS	TXS	TSSSXS
TSSXXVV	TSSSXXVV	TXXVPXVV
TXXTTVV	TSXXTVV	TSSXXVPS
TSXXTVPS	TXXTVPS	TXXVPS
PVV	PTTVV	PTVPXVV
PVPXVV	PTVPXTVV	PVPXVPS
PTVPS	PVPXTVPS	PTTTVPS

Figure 13
The 21 grammatical strings of length 3 to 8

• Subpatterns occurring in the strings are not all associated with their possible successors equally often. Accurate predictions therefore require that information about the identity of the letters that have already been presented be maintained in the system, i.e., the system must be sensitive to the frequency distribution of subpatterns in the training set. This

amounts to encoding the full path that has been traversed in the grammar.
These features of the limited training set obviously make the prediction task much more complex than in the previous simulation.

3.2 Network Architecture

The same general network architecture was used for this set of simulations. The number of hidden units was arbitrarily set to 15.

3.3 Performance

The network was trained on the 22 different sequences (a total of 130 patterns) until the total sum squared error (*tss*) reached a plateau with no further improvements. This point was reached after 2000 epochs and *tss* was 50. Note that *tss* cannot be driven much below this value, since most partial sequences of letters are compatible with 2 different successors. At this point, the network correctly predicts the possible successors of each letter, and distinguishes between different occurrences of the same letter -- like it did in the simulation described previously. However, the network's performance makes it obvious that many additional constraints specific to the limited training set have been encoded. Figure 14 shows that the network expects a 'T' or a 'V' after a first presentation of the second 'X' in the grammar.

```
string     btxXvpxvv

            B  T  S  P  X  V  E
output      0 49  0  0  0 50  0

hidden     27 89  2 16 99 43  1  6  4 18 99 81 95 18  1

context     1 18  0 41 95  1 60 59  5  6 84 99 19  5  0

            B  T  S  P  X  V  E
input       0  0  0  0100  0  0
```

Figure 14
State of the network after presentation of the second 'X'

Contrast these predictions with those illustrated in Figure 15, which shows the state of the network after a *second* presentation of the second 'X' : Although the same node in the grammar has been reached, and 'T' and 'V' are again possible alternatives, the network now predicts only 'V'.

```
string     btxxvpXvv

            B  T  S  P  X  V  E
output     0  3  0  0  0 95  0

hidden     0 85  3 85 31  0 72 19 31  3 93 99 61  5  0

context    1  7  5 90 93  4  0 10 71 40 99 16 90  5 82

            B  T  S  P  X  V  E
inputn      0  0  0  0100  0  0
```

Figure 15

State of the network after a second presentation of the second 'X'

Thus, the network has successfully learned that an 'X' occurring late in the sequence is never followed by a 'T'-- a fact which derives directly from the maximum length constraint of 8 letters.

It could be argued that the network simply learned that when 'X' is preceded by 'P' it cannot be followed by 'T', and thus relies only on the preceding letter to make that distinction. However, the story is more complicated than this.

In the following two cases, the network is presented with the first occurrence of the letter 'V'. In the first case, 'V' is preceded by the sequence 'tssxx', while in the second case, it is preceded by 'tsssxx'. The difference of a single 'S' in the sequence--which occurred 5 presentations before--results in markedly different predictions when 'V' is presented (Figure 16).

The difference in predictions can be traced again to the length constraint imposed on the strings in the limited training set. In the second case, the string spans a total of 7 letters when 'V' is presented, and the only alternative compatible with the length constraint is a second 'V' and the end of the string. This is not true in the first case, in which both 'VV' and 'VPS' are possible endings.

Thus, it seems that the representation developed on the context units encodes more than the immediate context -- the pattern of activation could include a full representation of the path traversed so far. Alternatively, it could be hypothesized that the context units encode only the preceding letter and a counter of how many letters have been presented.

In order to understand better the kind of representations that evolve on the context units we performed a cluster analysis on all the hidden unit patterns evoked by each sequence. Each letter of each sequence was presented to the network and the corresponding pattern of activation on the hidden layer was recorded. The Euclidean distance between each pair of patterns was computed and the matrix of all distances was provided as input to a cluster analysis program.

```
string     btssxxVv

           B  T  S  P  X  V  E
output     0  0  0 54  0 48  0

hidden    44 98 30 84 99 82  0 47  0  9 41 98 13  2  0

context   89 90  1  1 99 70  1  3  2 10 99 95 85 21  0

           B  T  S  P  X  V  E
input      0  0  0  0  0100  0


string     btsssxxVv

           B  T  S  P  X  V  E
output     0  0  0  2  0 97  0

hidden    56 99 48 93 99 85  0 22  0 10 77 97 30  3  0

context   54 67  1  4 99 59  7  9  1  6 98 97 72 16  0

           B  T  S  P  X  V  E
input      0  0  0  0  0100  0
```

Figure 16
Two presentations of the first 'V', with slightly different paths

The resulting analysis is shown in Figure 17. We labeled the arcs according to the letter being presented (the 'current letter') and its position in the grammar defined by Reber. Thus 'V_1' refers to the first 'V' in the grammar and 'V_2' to the second 'V' which immediately precedes the end of the string. 'Early' and 'Late' refer to whether the letter occurred early or late in the sequence (for example in 'PT..' 'T_2' occurs early; in 'PVPXT..' it occurs late). Finally, in the left margin we indicated what predictions the corresponding patterns yield on the output layer (e.g, the hidden unit pattern generated by 'BEGIN' predicts 'T' or 'P').

From the figure, it can be seen that the patterns are grouped according to three distinct principles: (1) according to similar predictions, (2) according to similar letters presented on the input units, and (3) according to similar paths. These factors do not necessarily overlap since several occurrences of the same letter in a sequence usually implies different predictions and since similar paths also lead to different predictions depending on the current letter.

For example, the top cluster in the figure corresponds to all occurrences of the letter 'V' and is further subdivided among 'V_1' and 'V_2'. The 'V_1' cluster is itself further divided

between groups where 'V_1' occurs early in the sequence (e.g, 'pV...') and groups where it occurs later (e.g, 'tssxxV...' and 'pvpxV...'). Note that the division according to the path does not necessarily correspond to different predictions. For example, 'V_2' always predicts 'END' and always with maximum certainty. Nevertheless, sequences up to 'V_2' are divided according to the path traversed.

Without going into the details of the organization of the remaining clusters, it can be seen that they are predominantly grouped according to the predictions associated with the corresponding portion of the sequence and then further divided according to the path traversed up to that point. For example, 'T_2', 'X_2' and 'P_1' all predict 'T or V', 'T_1' and 'X_1' both predict 'X or S', and so on.

Overall, the hidden units patterns developed by the network reflect two influences: a 'top-down' pressure to produce the correct output, and a 'bottom-up' pressure from the successive letters in the path which modifies the activation pattern independently of the output to be generated.

The top-down force derives directly from the back-propagation learning rule. Similar patterns on the output units tend to be associated with similar patterns on the hidden units. Thus, when two different letters yield the same prediction (e.g, 'T_1' and 'X_1'), they tend to produce similar hidden layer patterns. The bottom-up force comes from the fact that, nevertheless, each letter presented with a particular context can produce a characteristic 'mark' on the hidden unit pattern. The hidden unit patterns are not truly an 'encoding' of the input, as is often suggested, but rather an encoding of the *association* between a particular input and the relevant prediction. It really reflects an influence from both sides.

Finally, it is worth noting that the very specific internal representations acquired by the network are nonetheless sufficiently abstract to ensure good generalization. We tested the network on the remaining untrained 22 strings of length 3 to 8 that can be generated by the grammar. Over the 165 predictions of successors in these strings, the network made an incorrect prediction (activation of an incorrect successor > 0.05) in only 10 cases, and it predicted only one of two continuations consistent with the grammar and length constraints in 10 other cases.

3.4 Is the simple recurrent network still a finite state machine?

In the previous sections, we have examined how the recurrent network encodes and uses information about meaningful subsequences of events, giving it the capacity to yield different outputs according to some specific traversed path, or to the length of strings. However, the network does not use a separate and explicit representation for non-local properties of the strings such as length. It only learns to associate different predictions to a subset of states; those that are associated with a more restricted choice of successors. Again there are no stacks or registers, and each different prediction is associated to a specific state on the context units. In that sense, the recurrent network that has learned to master this task still behaves like a finite-state machine, although the training set involves non-local constraints that could only be encoded in a very cumbersome way in a finite-state *grammar*.

We usually do not think of finite state automata as capable of encoding non-local information such as length of a sequence. Yet, finite state machines have in principle the same computational power as a Turing machine with a finite tape and they can be designed

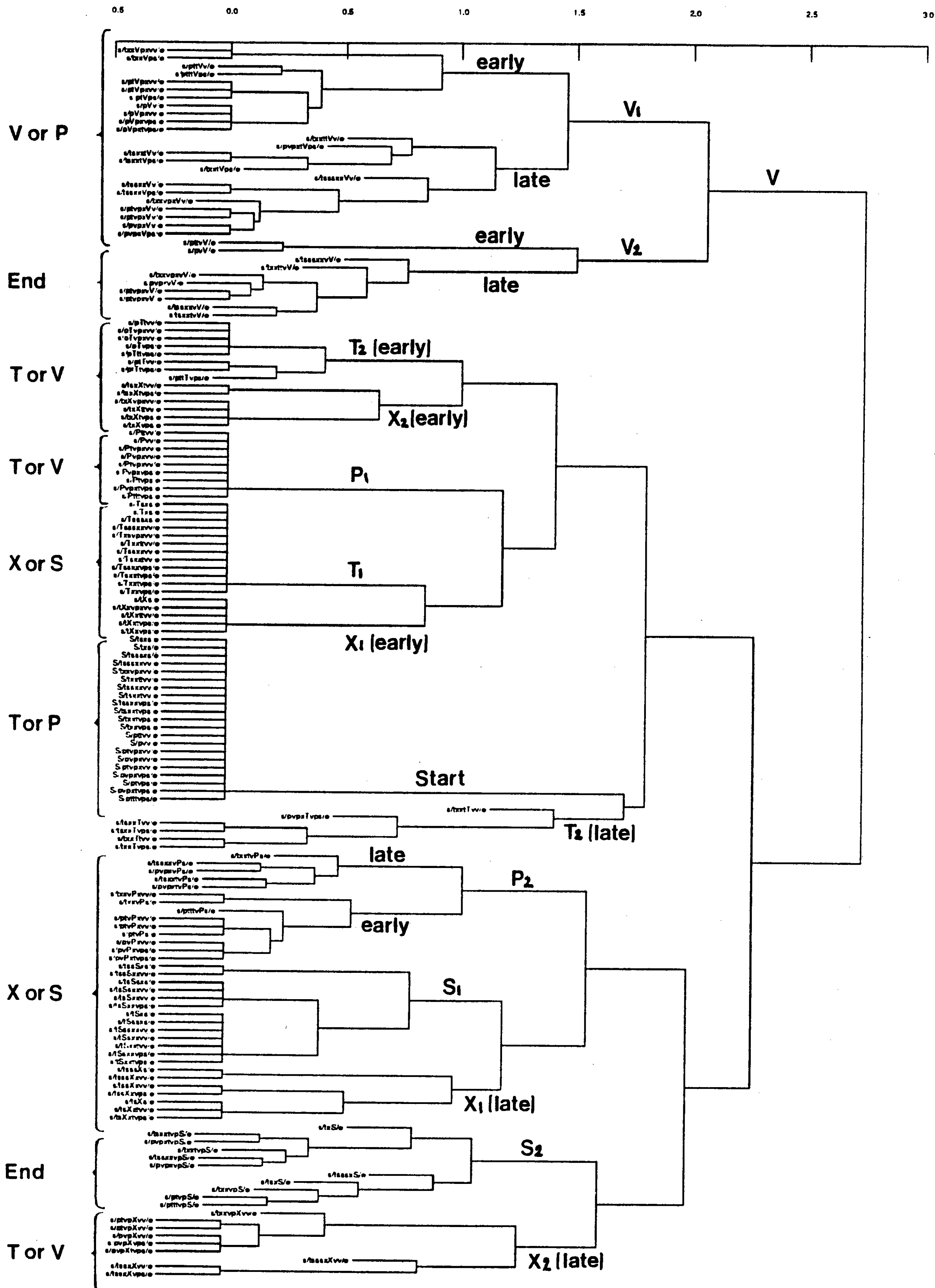

Figure 17
Hierarchical Cluster Analysis of the H.U. activation patterns after 2000 epochs of training on the 22 strings

to respond adequately to non-local constraints. Augmented transition networks and other Turing-equivalent automata are preferable to finite state machines because they spare memory and are modular -- and therefore easier to design and modify. However, the finite state machines that the recurrent network seems to implement have properties that set them apart from their traditional counterparts:

• For tasks with an appropriate structure, recurrent networks develop their own state transition diagram, sparing this burden to the designer.

• The large amount of memory required to develop different representations for every state needed is provided by the representational power of hidden layer patterns. For example, 15 hidden units with four possible values (say, 0, .25, .75, 1) can support more than one billion different patterns.

• The network implementation remains capable of performing similarity-based processing, making it somewhat noise-tolerant (the machine does not 'jam' if it encounters an undefined state transition and it can recover as the sequence of inputs continues), and it remains able to generalize to sequences that were not part of the training set.

In the next section, we examine how the simple recurrent network comes to develop its internal representations.

4. Learning

We have seen that the simple recurrent network develops and learns to use a compact and precise representation of the sequences presented. This representation is sufficient to disambiguate identical cues in the presence of context, to code for length constraints and to react appropriately to atypical cases[4]. How is this representation discovered?

To clarify how the network learns to use the context of preceding letters in a sequence, we will illustrate the different phases of learning with cluster analyses of the hidden layer patterns generated at each phase. To make the analyses simpler, we used a smaller training set than the training set mentioned previously. The corresponding finite-state grammar is shown in Figure 18.

In this simpler grammar, the main differences -- besides the reduced number of patterns -- are that the letters 'P' and 'T' appear only once and that the 'BEGIN' signal should now produce similar predictions as 'S_1', 'X_1' and 'P' (i.e, 'X or S')[5].

[4] In fact, length constraints are treated exactly as atypical cases since there is no representation of the length of the string as such.

[5] The actual training set we used can easily be obtained by removing all the strings beginning with 'P' in the full training set illustrated in Figure 12 and by removing the inital 'T' from the remaining strings.

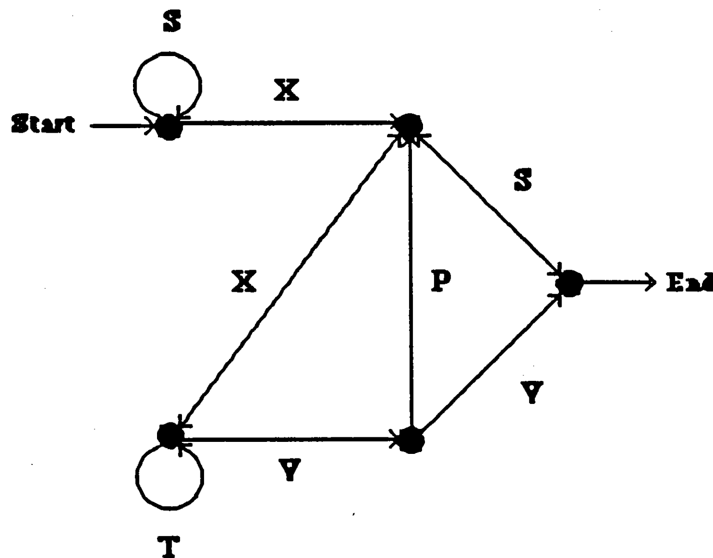

Figure 18
The reduced finite-state grammar

4.1 Discovering letters

At epoch 0, before the network has received any training, the hidden unit patterns clearly show an organization by letter: to each letter corresponds an individual cluster. Interestingly, these clusters are already subdivided according to preceding sequences -- the 'path'. This fact nicely illustrates how a pattern of activation on the context units naturally tends to encode the path traversed so far independently of any error correcting procedure. The average distance between the different patterns -- the 'contrast' as it were -- is nonetheless rather small; the scale only goes up to 0.6 [6] (see Figure 19). But this is due to the very small initial random values of the weights from the input and context layers to the hidden layer. Larger initial values would enhance the network's tendency to capture path information in the hidden unit patterns before training is even started.

After 100 epochs of training, an organization by letters is still prevalent, however letters have been regrouped according to similar predictions. 'START', 'P' and 'S' all make the common prediction of 'X or S' (although 'S' also predicts 'END'); 'T' and 'V' make the common prediction of 'V' (although 'V' also predicts 'END' and 'P'). The path information has been almost eliminated: there is very little difference between the patterns generated by two different occurrences of the same letter (see Figure 20). For example, the pattern generated by 'S_1' and the corresponding output are almost identical to the patterns generated by 'S_2', as Figures 21 and 22 demonstrate.

---

6 In all the following figures, the scale was automatically determined by the cluster analysis program. It is important to keep this in mind when comparing the figures to each other.

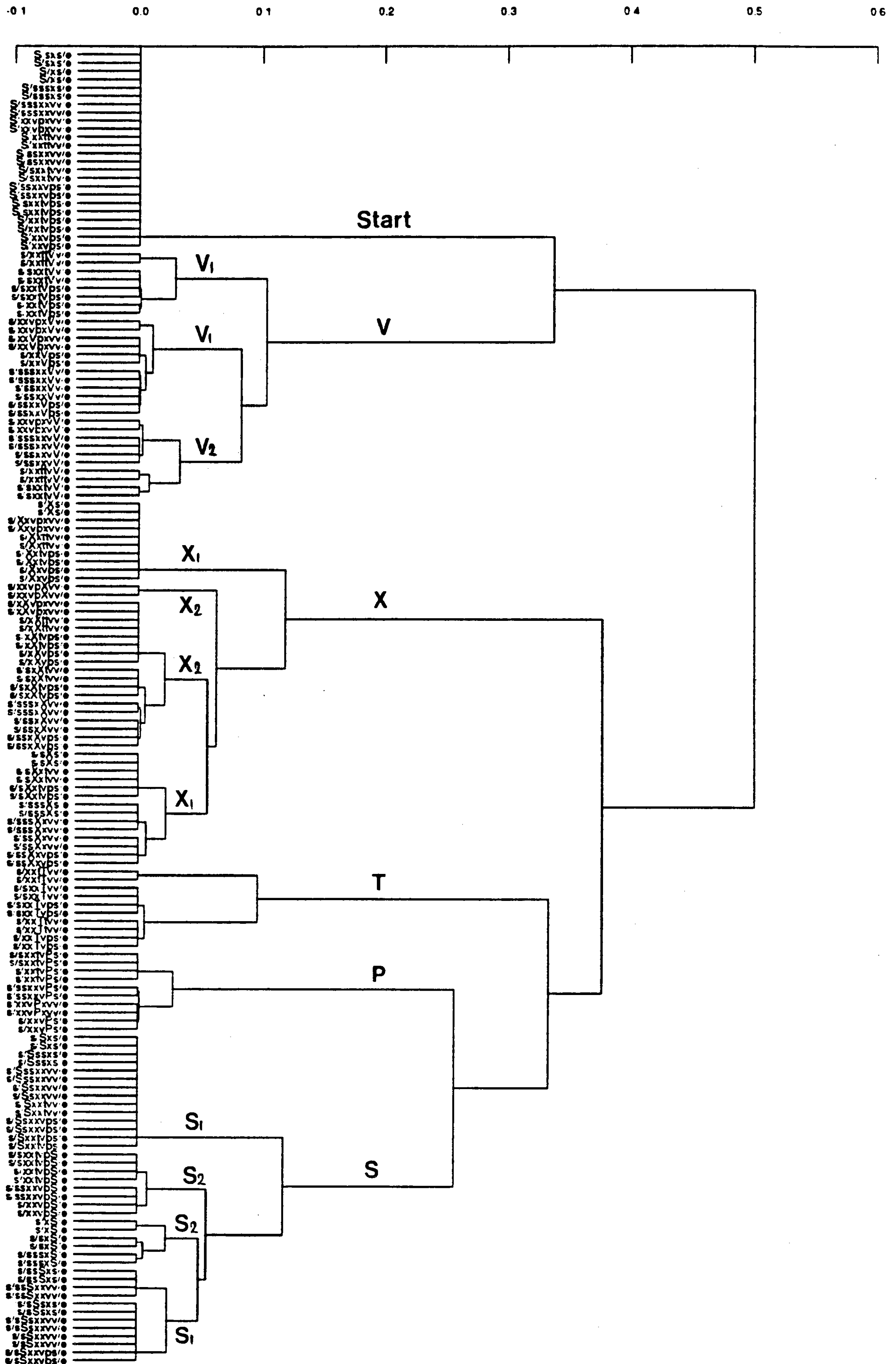

Figure 19
Hierarchical Cluster Analysis of the H.U. activation patterns before training (reduced set of strings)

-0.2 0.0 0.2 0.4 0.6 0.8 1.0

Start

P

S_2

S_2

S

S_1

S_1

X_2

X

X_1

T

V_1

V

V_1

V_2

Figure 20
Hierarchical Cluster Analysis of the H.U. activation patterns after
100 epochs of training (reduced set of strings)

```
epoch      100
string     bSsxxvps

            B  T  S   P  X  V  E
output      0  0 36   0 33 16 17

hidden     45 24 47 26 36 23 55 22 22 26 22 23 30 30 33

context    44 22 56 21 36 22 64 16 13 23 20 16 25 21 40

            B  T  S   P  X  V  E
input       0  0100   0  0  0  0
```

Figure 21
Hidden layer and output patterns generated by the presentation of the first S in a sequence after 100 epochs of training

The network is learning to 'ignore' the pattern of activation on the context units and to produce an output pattern appropriate to the letter 'S' in any context. This is a direct consequence of the fact that the patterns of activation on the hidden layer -- and hence the context layer -- are continuously changing from one epoch to the next as the weights from the input units (the letters) to the hidden layer are modified. Consequently, adjustments made to the weights from the context layer to the hidden layer are inconsistent from epoch to epoch and cancel each other. In contrast, the network is able to pick up the stable association between each letter and all of its possible successors.

```
epoch      100
string     bssxxvpS

            B  T  S   P  X  V  E
output      0  0 37   0 33 16 17

hidden     45 24 47 25 36 23 56 22 21 25 21 22 29 30 32

context    42 29 53 24 32 27 61 25 16 33 25 23 28 27 41

            B  T  S   P  X  V  E
input       0  0100   0  0  0  0
```

Figure 22
Hidden layer and output patterns generated by the presentation of the second S in a sequence after 100 epochs of training

4.2 Discovering arcs

At the end of this phase, individual letters consistently generate a unique pattern of activation on the hidden layer. This is a crucial step in developing a sensitivity to context: patterns copied onto the context layer have become a unique code designating which letter immediately preceded the current letter. The learning procedure can now exploit the regular association between the pattern on the context layer and the desired output. Around epoch 700, the cluster analysis shows that the network has used this information to differentiate clearly between the first and second occurrence of the same letter (Figure 23). The pattern generated by 'S_2' -- which predicts 'END' -- clusters with the pattern generated by 'V_2', which also predicts 'END'. The overall difference between all the hidden layer patterns has also more than roughly doubled, as indicated by the change in scale.

4.3 Encoding the path

During the last phase of learning, the network learns to make different predictions to the same occurrence of a letter (e.g, 'V_1') on the basis of the previous sequence. For example, it learns to differentiate between 'ssxxV' which predicts either 'P' or 'V', and 'sssxxV' which predicts only 'V'. As we discussed previously, in order to make this distinction, the pattern of activation on the context layer must be a representation of the entire path rather than simply an encoding of the previous letter. The fact that discovering the path is the last phase in learning suggests that it is the hardest one. In the following section, we propose a detailed analysis of the constraints guiding the learning of complex contingencies requiring information to be maintained for several processing steps.

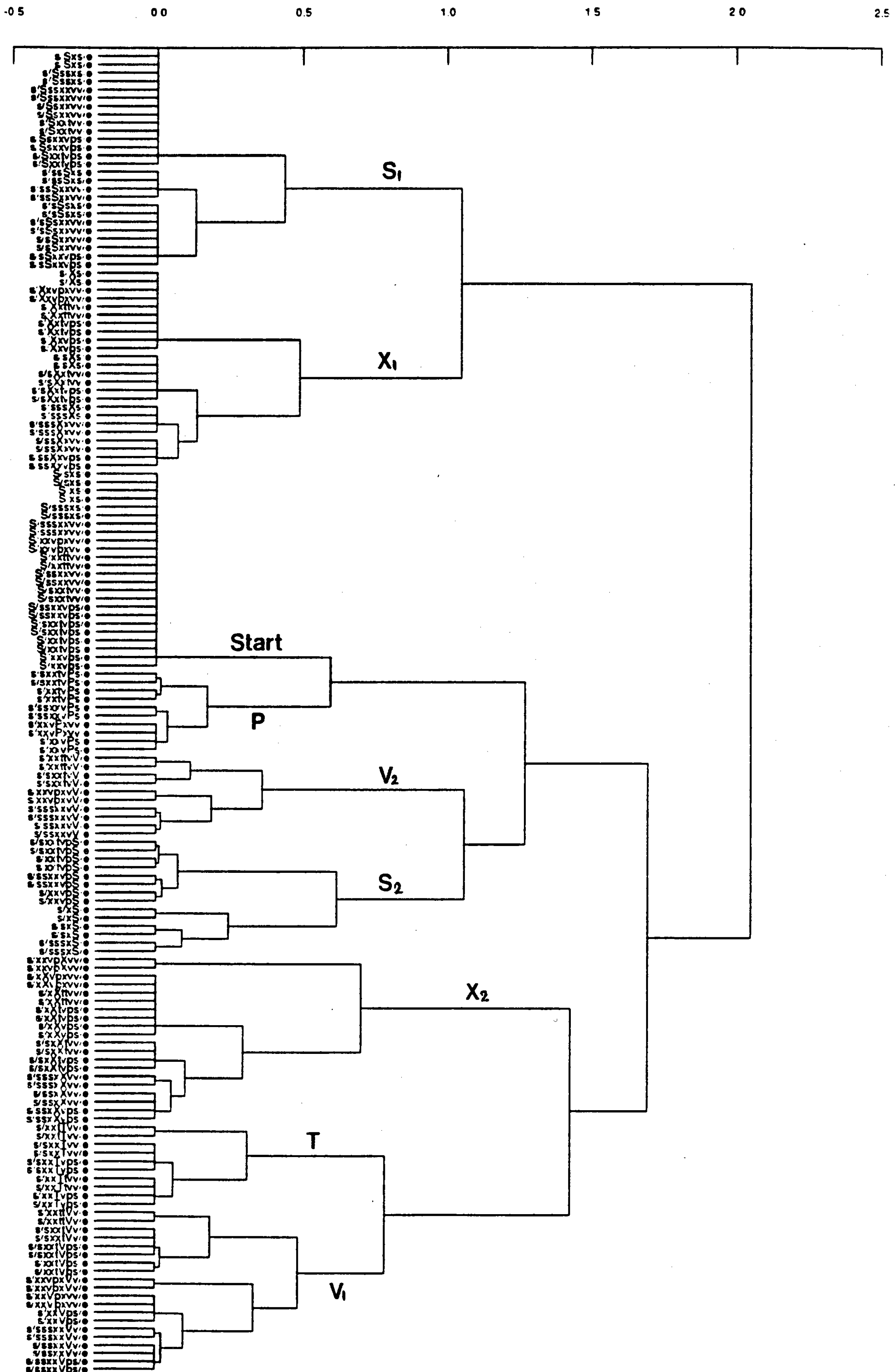

Figure 23
Hierarchical Cluster Analysis of the H.U. activation patterns after
700 epochs of training (reduced set of strings)

5. Encoding non-local context

5.1 Processing loops

Consider the general problem of learning two arbitrary sequences of the same length and ending by two different letters. Under what conditions will the network be able to make a correct prediction about the nature of the last letter when presented with the penultimate one? Obviously, the *necessary and sufficient condition* is that the internal representations associated with the penultimate letter are different (Indeed, the hidden units patterns *have* to be different if different outputs are to be generated.). Let us consider several different prototypical cases and verify if this condition holds :

PABC X and PABC V [1]

PABC X and TDEF V [2]

Clearly, problem [1] is impossible : as the two sequences are identical up to the last letter, there is simply no way for the network to make a different prediction when presented with the penultimate letter ('C' in the above example). The internal representations induced by the successive elements of the sequences will be strictly identical in both cases. Problem [2], on the other hand, is trivial, as the last letter is contingent on the penultimate letter ('X' is contingent on 'C'; 'V' on 'F'). There is no need here to maintain information available for several processing steps, and the different contexts set by the penultimate letters are sufficient to ensure that different predictions can be made for the last letter. Consider now problem [3].

PSSS P and TSSS T [3]

As can be seen, the presence of a final 'T' is contingent on the presence of an initial 'T'; a final 'P' on the presence of an initial 'P'. The shared 'S's do not supply any relevant information for disambiguating the last letter. Moreover, the predictions the network is required to make in the course of processing are identical in both sequences up to the last letter.

It can easily be shown that the only way for the network to solve this problem is to develop *different internal representations* for *every* letter in the two sequences.

First, consider the fact that the network is required to make different predictions when presented with the last 'S'. As stated earlier, this will only be possible if the input presented at the penultimate time step produces different internal representations in the two sequences. However, this necessary difference can not be due to the last 'S' itself, as it is presented in both sequences. Rather, the only way for different internal representations to arise when the last 'S' is presented is when the context pool holds different patterns of activation. As the context pool holds a copy of the internal representations of the previous step, these representations must themselves be different. Recursively, we can apply the same reasoning up to the first letter. The network must therefore develop a different representation for all the letters in the sequence. Are initial different letters a sufficient

condition to ensure that each letter in the sequences will be associated with different internal representations? The answer is twofold.

First, note that developing a different internal representation for each letter (including the different instances of the letter 'S') is provided *automatically* by the recurrent nature of the architecture, even without any training. Successive presentations of identical elements to a recurrent network generate different internal representations at each step because the context pool holds different patterns of activity at each step. In the above example, the first letters will generate different internal representations. On the following step, these patterns of activity will be fed back to the network, and induce different internal representations again. This process will repeat itself up to the last 'S', and the network will therefore find itself in a state in which it is potentially able to correctly predict the last letter of the two sequences of problem [3]. Now, there is an important caveat to this observation. Another fundamental property of recurrent networks is convergence towards an attractor state when a long sequence of identical elements are presented. Even though, initially, different patterns of activation are produced on the hidden layer for each 'S' in a sequence of 'S's, eventually the network converges towards a stable state in which every new presentation of the same input produces the same pattern of activation on the hidden layer. The number of iterations required for the network to converge depends on the number of hidden units. With more degrees of freedom, it takes more iterations for the network to settle. Thus, increasing the number of hidden units provides the network with an increased *architectural* capacity of maintaining differences in its internal representations when the input elements are identical.[7]

Second, consider the way back-propagation interacts with this natural process of maintaining information about the first letter. In problem [3], the predictions in each sequence are identical up to the last letter. As similar outputs are required on each time step, the weight adjustment procedure pushes the network into developing *identical* internal representations at each time step and for the two sequences - therefore going in the opposite direction than is required. This 'homogenizing' process can strongly hinder learning, as will be illustrated below.

From the above reasoning, we can infer that optimal learning conditions exist when both contexts and predictions are different in each sequence. If the sequences share identical sequences of predictions -- as in problem [3] -- the process of maintaining the differences between the internal representations generated by an (initial) letter can be disrupted by back-propagation itself. The very process of learning to predict correctly the intermediate shared elements of the sequence can even cause the total error to rise sharply in some cases after an initial decrease. Indeed, the more training the network gets on these intermediate elements, the more likely it is that their internal representations will become identical, thereby completely eliminating initial slight differences that could potentially were being used to disambiguate the last element. Further training can only worsen this situation.[8]

[7] For example, with three hidden units, the network converges to a stable state after an average of three iterations when presented with identical inputs (with a precision of two decimal points for each unit). A network with 15 hidden units converges after an average of 8 iterations. These results were obtained with random weights in the range [-0.5,+0.5].

[8] Generally, small values for the learning rate and momentum, as well as many hidden units, help to minimize this problem.

Note that in this sense back-propagation in the recurrent network is not guaranteed to implement gradient descent. Presumably, the ability of the network to resist the 'homogenization' induced by the learning algorithm will depend on its representational power - the number of hidden units available for processing. With more hidden units, there is also less pressure on each unit to take on specified activation levels. Small but crucial differences in activation levels will therefore be allowed to survive at each time step, until they finally become useful at the penultimate step.

To illustrate this point, a network with fifteen hidden units was trained on the two sequences of problem [3]. The network is able to solve this problem very accurately after approximately 10,000 epochs of training on the two patterns. Learning proceeds smoothly until a very long plateau in the error is reached. This plateau corresponds to a learning phase during which the weights are adjusted so that the network can take advantage of the small differences that remain in the representations induced by the last 'S' in the two strings in order to make accurate predictions about the identity of the last letter. These slight differences are of course due to the different context generated after presentation of the first letter of the string.

To understand further the relation between network size and problem size, four different networks (with 7, 15, 30 or 120 hidden units) were trained on each of four different versions of problem [3] (with 2, 4, 6 or 12 intermediate elements). As predicted, learning was faster when the number of hidden units was larger. There was an interaction between the size of the network and the size of the problem : adding more hidden units was of little influence when the problem was small, but had a much larger impact for larger numbers of intervening 'S's. We also observed that the relation between the size of the problem and the number of epochs to reach a learning criterion was exponential for all network sizes. These results suggest that for relatively short embedded sequences of identical letters, the difficulties encountered by the simple recurrent network can be alleviated by increasing the number of hidden units. However, beyond a certain range, maintaining different representations across the embedded sequence becomes exponentially difficult.

An altogether different approach to the question can also be taken. In the next section, we argue that some sequential problems may be less difficult than problem [3]. More precisely, we will show how very slight adjustments to the predictions the network is required to make in otherwise identical sequences can greatly enhance performance.

5.2 Spanning embedded sequences

The previous example is a limited test of the network's ability to preserve information during processing of an embedded sequence in several respects. Relevant information for making a prediction about the nature of the last letter is at a constant distance across all patterns; elements inside the embedded sequence are all identical; and most importantly, the nature of predictions inside the embedded sequence is independent of the predecessor of the embedding -- the element that determines what assignment should be made upon exit from the embedded sequence.

These three factors make the task of the network radically different from a similar prediction task that would be applied to natural language. Moreover, the fact that contingencies inside the embedded sequence are similar in all the training exemplars greatly raises the difficulty of the task.

Consider a problem of number agreement illustrated by the following two sentences:

The **dog** *that chased the cat* **is** very playful
The **dogs** *that chased the cat* **are** very playful

We would contend that expectations about concepts and words forthcoming in the embedded sentence are different for the singular and plural forms. For example, the embedded clauses require different agreement morphemes -- chases vs. chase -- when the clause is in the present tense, etc. Furthermore, even after the same word has been encountered in both cases (e.g, 'chased'), expectations about possible successors for that word would remain different. (e.g, a single dog and a pack of dogs are likely to be chasing different things). As we have seen, if such differences in predictions do exist the network is more likely to maintain information relevant to non-local context since that information is relevant at several intermediate steps.

To illustrate this point, the network was trained on the finite state grammar shown in Figure 24. If the first letter encountered in the string is a 'T', the last letter of the string is also a 'T'. Conversely, if the first letter is a 'P', the last letter is also a 'P'. In between these matching letters, we interposed the same finite state grammar that we had been using in previous experiments (Reber's) to play the role of an embedded sentence. We modified Reber's grammar slightly by making the probability of traversing the 'S' loop and the 'T' loop only 0.3 rather than 0.5 in order to shorten the average length of strings.

In a first experiment we trained the network on strings generated from the finite state grammar with the same probabilities attached to corresponding arcs in the bottom and top version of Reber's grammar. This version was called the 'symmetrical grammar': contingencies inside the sub-grammar are the same independently of the first letter of the string, and all arcs except for the 'S' and the 'T' loop and their complements have a probability of 0.5. In a second experiment the network was trained on an 'asymmetrical' version in which the top sub-grammar was slightly biased towards the top nodes (probability of the first 'T' was 0.6 vs. 0.4 for the first 'P'; 0. 6 for the second 'S' vs. 0.4 for the second 'X', 0.6 for the second 'P' vs. 0.4 for the second 'V'). Conversely, probabilities in the bottom sub-grammar were biased in the opposite direction. For both the symmetrical and asymmetrical grammars, the average length of strings was 7 with a standard deviation of 2.6.

After training, the performance of each network was evaluated in the following way : 20,000 strings *generated from the symmetrical grammar* were presented and for each string we looked at the relative activation of the predictions of 'T' and 'P' upon exit from the sub-grammar. If the Luce ratio for the prediction with the highest activation was below 0.6, the trial was treated as a 'miss' (i.e, failure to predict one or the other distinctively).[9]

---

[9] The Luce ratio is the ratio of the highest activation on the output layer to the sum of all activations on that layer. This measure is commonly applied in psychology to model the strength of a response tendency among a finite set of alternatives. In this simulation, a Luce ratio of 0.5 often corresponded to a situation where 'T' and 'P' were equally activated and all other alternatives were set to zero.

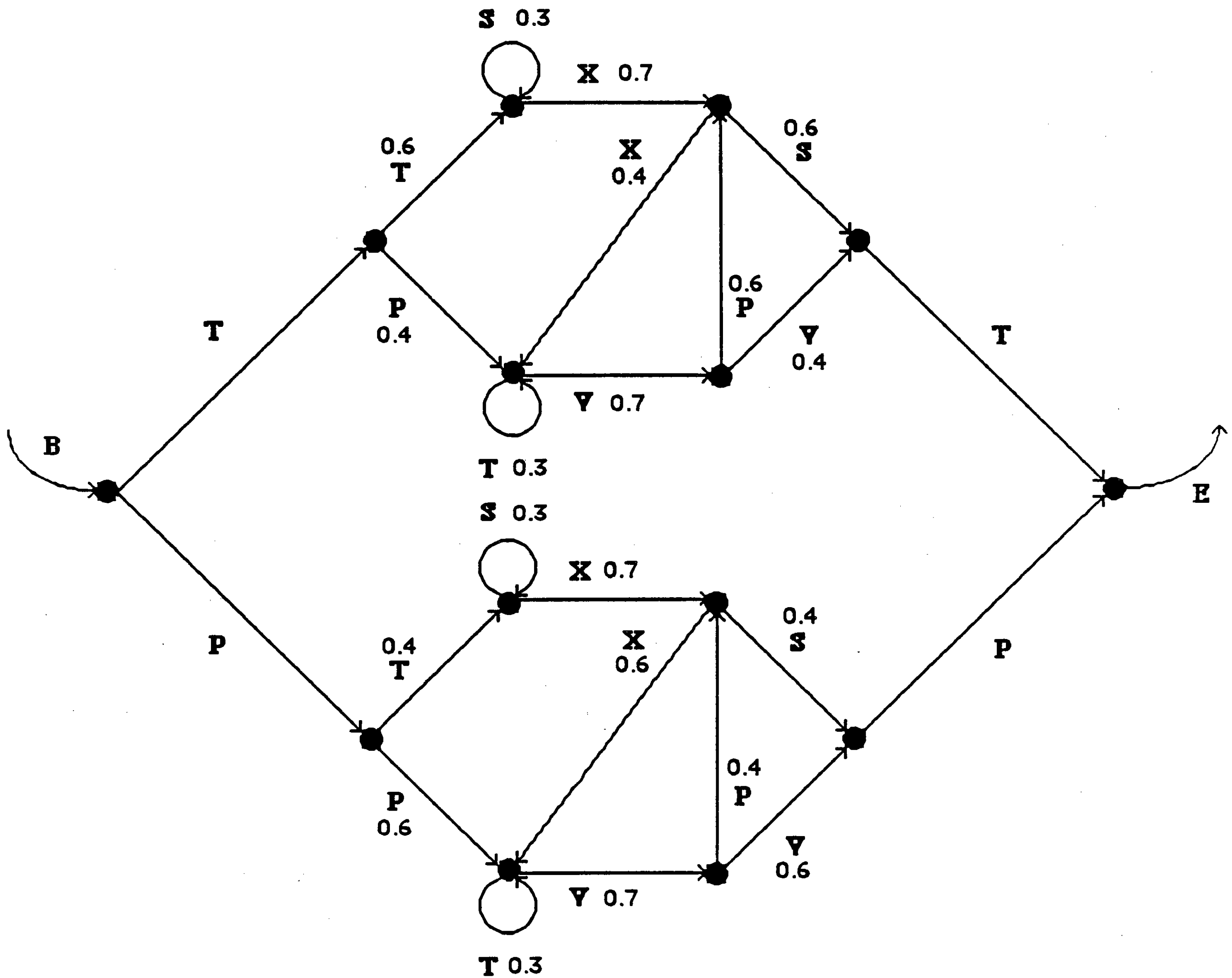

Figure 24
A complex finite-state grammar involving an embedded clause. The last letter is contingent on the first one, and the intermediate structure is shared by the two branches of the grammar. Arcs in the asymmetrical version have different transitional probabilities in the top and bottom sub-structure as explained in the text.

If the Luce ratio was greater or equal to 0.6 and the network predicted the correct alternative, a 'hit' was recorded. If the incorrect alternative was predicted, the trial was

treated as an 'error'.

Figure 25 gives a graphic summary of the performance of the network trained on the symmetrical grammar. After 2.4 million presentations, this network was unable to predict the correct alternative above threshold.

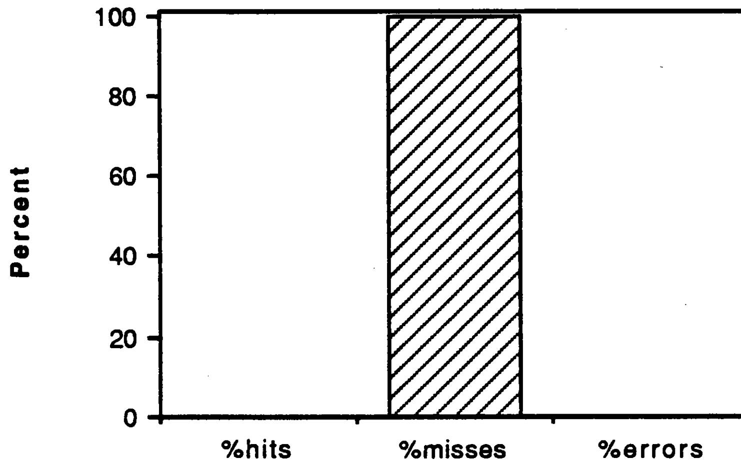

Figure 25
Percentage of hits, misses and errors in the network's predictions about the identity of the last letter following training on the symmetrical grammar

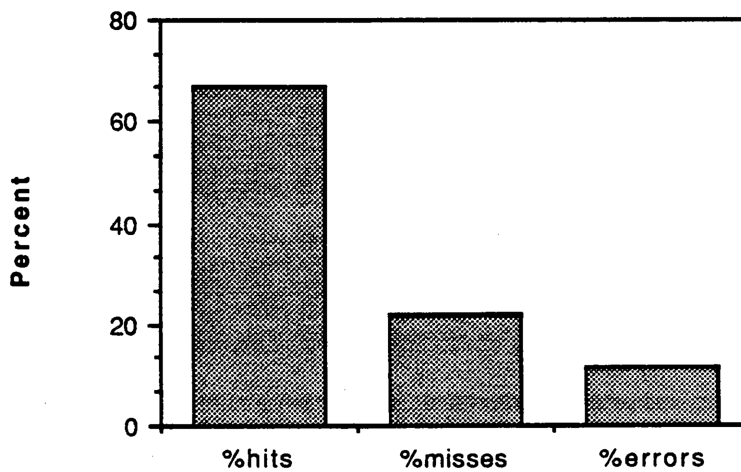

Figure 26
Percentage of hits, misses and errors in the network's predictions about the identity of the last letter following training on the asymmetrical grammar

Figure 26 shows the performance of the network trained on the *asymmetrical* version of the grammar. The correct prediction was generated in more than two thirds of the trials. Performance was best for shorter strings (i.e, three or four letters in the embedding) and deteriorated progressively as the length of the embedding increased (see figure 27). It is important to note that performance of this network cannot be attributed to a difference in statistical properties of the test strings between the top and bottom sub-grammars -- such as the difference present during training -- since the testing set came from the *symmetrical* grammar. Therefore, this experiment demonstrates that the network is able to preserve information about the predecessor of the embedded sequence across identical embeddings as long as the ensemble of *potential* pathways is differentiated during training. Furthermore, differences in potential pathways may be only statistical and, even then, rather small. Finally, the potential pathways do not have to differ at *every* step. For example, predictions within the 'T' or the 'S' loop are identical in both subgrammars in the asymmetrical training set.

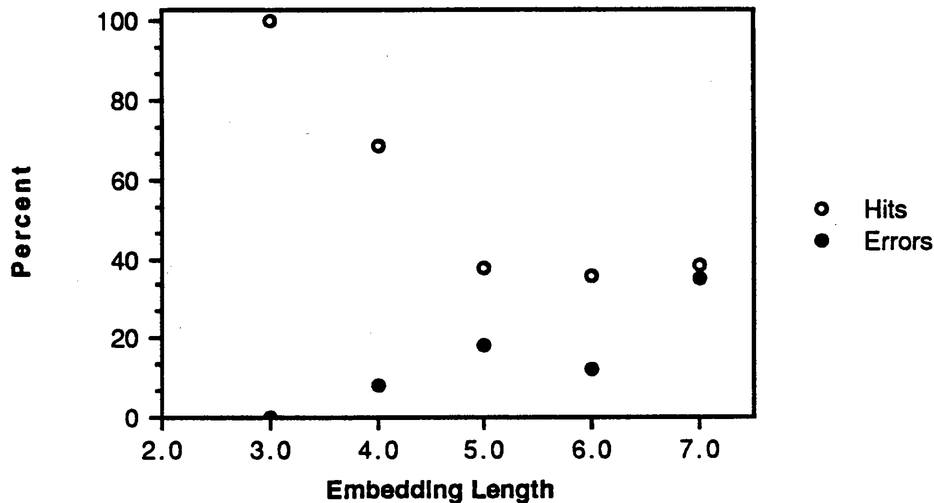

Figure 27
Percentage of hits and errors as a function of embedding length.
All the cases with 7 or more letters in the embedding were grouped together.

6. Discussion

In this study, we attempted to understand better how the simple recurrent network could learn to represent and use contextual information when presented with structured sequences of inputs. Following the first experiment, we concluded that copying the state of activation on the hidden layer at the previous time step provided the network with the basic equipment of a finite state machine. When the set of exemplars that the network is trained on comes from a finite state grammar, the network can be used as a recognizer with respect to that

grammar. When the representational resources are severely constrained, internal representations actually converge on the nodes of the grammar. Interestingly, though, this representational convergence is not a necessary condition for functional convergence: networks with more than enough structure to handle the prediction task, sometimes represent the same node of the grammar using two quite different patterns, corresponding to different paths into the same node. This divergence of representations does not upset the network's ability to serve as a recognizer for well-formed sequences derived from the grammar.

We also showed that the mere presence of recurrent connections pushed the network to develop hidden layer patterns that capture information about sequences of inputs, even in the absence of training. The second experiment showed that back-propagation can be used to take advantage of this natural tendency when information about the path traversed is relevant to the task at hand. This was illustrated with predictions that were specific to particular subsequences in the training set or that took into account constraints on the length of sequences. Encoding of sequential structure depends on the fact that back-propagation causes hidden layers to encode task-relevant information. In the simple recurrent network, internal representations encode not only the prior event but also relevant aspects of the representation that was constructed in predicting the prior event from its predecessor. When fed back as input, these representations provide information that allows the network to maintain prediction-relevant features of an entire sequence. We illustrated this with cluster analyses of the hidden layer patterns.

Our description of the stages of learning suggested that the network initially learns to distinguish between events independently of the temporal context (e.g, simply distinguish between different letters). The information contained in the context layer is ignored at this point. At the end of this stage, each event is associated to a specific pattern on the hidden layer that identifies it for the following event. In the next phase, thanks to this new information, different occurrences of the same event (e.g, two occurrences of the same letter) are distinguished on the basis of immediately preceding events -- the simplest form of a time tag. This stage corresponds to the recognition of the different 'arcs' in the particular finite state grammar used in the experiments. Finally, as the representation of each event progressively acquires a time tag, sub-sequences of events come to yield characteristic hidden layer patterns that can form the basis of further discriminations (e.g, between an 'early' and 'late' 'T_2' in the Reber grammar). In this manner, and under appropriate conditions, the hidden unit patterns achieve an encoding of the entire sequence of events presented.

We do not mean to suggest that simple recurrent networks can learn to recognize *any* finite state language. Indeed, we were able to predict two conditions under which performance of the simple recurrent network will deteriorate: (1) when different sequences may contain identical embedded sequences involving *exactly* the same predictions; and (2) when the number of hidden units is restricted and cannot support redundant representations of similar predictions, so that identical predictions following different events tend to be associated with very similar hidden unit patterns thereby erasing information about the initial path. We also noted that when recurrent connections are added to a three-layer feed-forward network, back-propagation is no longer guaranteed to perform gradient descent in the error space. Additional training, by improving performance on shared components of otherwise differing sequences, can eliminate information necessary to 'span' an embedded sequence and result in a sudden rise in the total error.

It follows from these limitations that the simple recurrent network could not be expected to

learn sequences with a moderately complex recursive structure -- such as context free grammars -- if contingencies inside the embedded structures do not depend on relevant information preceding the embeddings.

What is the relevance of this work with regard to language processing ? Obviously, the prediction paradigm is only remotely related to the very complex requirements of language processing. However, there is little doubt that language users develop expectations about the next elements in a linguistic event, and Elman (1988) has shown that simple recurrent networks that develop such expectations acquire representations that capture grammatical category information as well as more subtle word-specific information about a word string. Further, StJohn and McClelland (1988) demonstrated that linguistic operations such as thematic role attributions can be approached by training a recurrent network to activate representations of appropriate role-filler pairs as it sees word sequences that form simple sentences. The present work has defined more precisely the type of tasks that an architecture as basic as the simple recurrent network can be expected to master successfully. In particular, we showed that very slight modifications in the probability structure of the grammar defining the material to be learned greatly enhances the quality of predictions. As we noted previously, natural linguistic stimuli may show this property. If this were the case, the performance of the recurrent network would shed a new light on language processing, suggesting that a significant amount of sequential context can be captured and used by very simple computational devices.

References

Cottrell, G.W. (1985). Connectionist Parsing. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum.

Elman, J.L. (1988). Finding structure in time. CRL Technical Report 8801. Center for Research in Language, University of California, San Diego.

Fanty, M. (1985). Context-Free Parsing in Connectionist Networks. TR174, Rochester, N.Y. : University of Rochester, Computer Science Department.

Hanson, S. & Kegl, J. (1987). PARSNIP : A connectionist network that learns natural language from exposure to natural language sentences. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum.

Hinton, G. (1986). Learning Distributed Representations of Concepts. In *Proceedings of the Eighth Annual Conference of The Cognitive Science Society*. Hillsdale, NJ: Erlbaum.

Jordan, M. I. (1986). Attractor Dynamics and Parallelism in a Connectionist Sequential Machine. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum.

McClelland, J.L., & Rumelhart, D.E. (1988) *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs and Exercises*. Cambridge, Mass : MIT Press.

Pollack, J. (1988). Recursive Auto-Associative Memory : devising Compositional Distributed Representations. Technical Report MCCS-88-124, Computing Research Laboratory, New Mexico State University.

Reber, A.S. (1967). Implicit learning of artificial grammars. *Journal of Verbal Learning and Verbal Behavior*, 5, 855-863.

Rumelhart, D.E., & McClelland, J.L. (1986). *Parallel Distributed Processing, I: Foundations*. Cambridge, Mass : MIT Press.

Rumelhart, D.E., Hinton, G., & Williams, R.J. (1986). Learning Internal Representations by Error Propagation. In Rumelhart, D.E. and McClelland, J.L. (Eds.), *Parallel Distributed Processing, I : Foundations*. Cambridge, Mass : MIT Press.

Sejnowski, T.J., & Rosenberg, C. (1986). NETtalk: A Parallel Network that Learns to Read Aloud. Technical Report, Johns Hopkins University JHU-EECS-86-01

StJohn, M., & McClelland, J.L. (1988). Learning and Applying Contextual Constraints in Sentence Comprehension. Technical Report, Department of Psychology, Carnegie Mellon University.