Machine Learning, 7, 161-193 (1991)
© 1991 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Graded State Machines: The Representation of Temporal Contingencies in Simple Recurrent Networks

DAVID SERVAN-SCHREIBER, AXEL CLEEREMANS AND JAMES L. McCLELLAND
School of Computer Science and Department of Psychology, Carnegie Mellon University

Abstract. We explore a network architecture introduced by Elman (1990) 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. Next, we provide a detailed analysis of how the network acquires its internal representations. We show that the network progressively encodes more and more temporal context by means of a probability analysis. Finally, 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. The network encodes long-distance dependencies by *shading* internal representations that are responsible for processing common embeddings in otherwise different sequences. This ability to represent simultaneously similarities and differences between several sequences relies on the graded nature of representations used by the network, which contrast with the finite states of traditional automata. For this reason, the network and other similar architectures may be called *Graded State Machines.*

Keywords. Graded state machines, finite state automata, recurrent networks, temporal contingencies, prediction task

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 behavior 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 in 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, 1987; 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 (1990) 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 sate over a set of 'state units' (Figure 1).

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 (1990) has introduced an architecture—which we call a simple recurrent network (SRN)—that has the potential to master an infinite corpus of sequences with the limited means of a learning procedure that is *completely local in time* (Figure 2). In the SRN, 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. These context units are comparable to Jordan's state units.

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

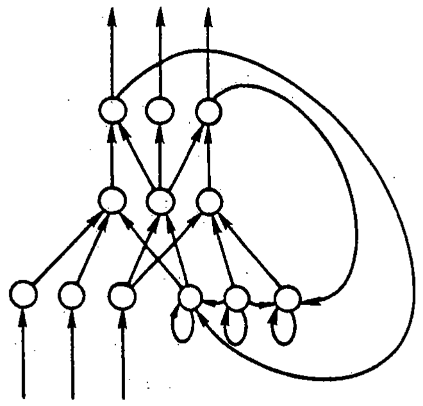

Figure 1. The Jordan (1986) Sequential Network.

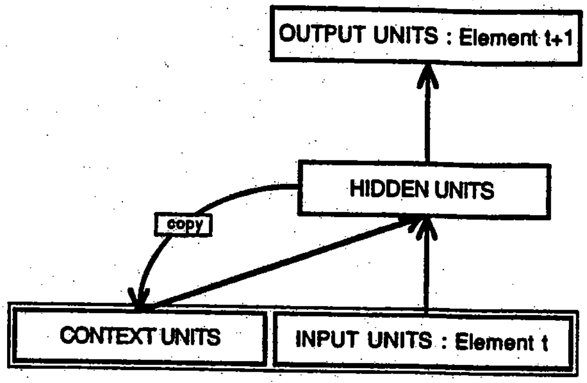

Figure 2. The Simple Recurrent Network. Each box represents a pool of units and each forward arrow represents a complete set of trainable connections from each sending unit to each receiving unit in the next pool. The backward arrow from the hidden layer to the context layer denotes a copy operation.

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., Hinton, McClelland & Rumelhart, 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.

In this study, we show that the SRN can learn to mimic closely a finite state automaton (FSA), both in its behavior and in its state representations. In particular, we show that it can learn to process an *infinite* corpus of strings based on experience with a *finite* set of training exemplars. We then explore the capacity of this architecture to recognize and use non-local contingencies between elements of a sequence that cannot be represented conveniently in a traditional finite state automaton. We show that the SRN encodes long-distance dependencies by *shading* internal representations that are responsible for processing common embeddings in otherwise different sequences. This ability to represent simultaneously similarities and differences between sequences in the same state of activation relies on the graded nature of representations used by the network, which contrast with the finite states of traditional automata. For this reason, we suggest that the SRN and other similar architectures may be exemplars of a new class of automata, one that we may call *Graded State Machines*.

2. Learning a finite state grammar

2.1. Material and task

In our first experiment, we asked whether the network cold learn the contingencies implied by a small finite state grammar. As 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 (1976) 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 current 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

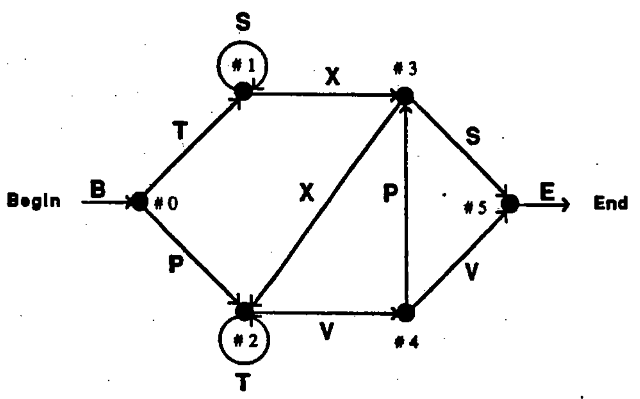

Figure 3. The finite-state grammar used by Reber (1976).

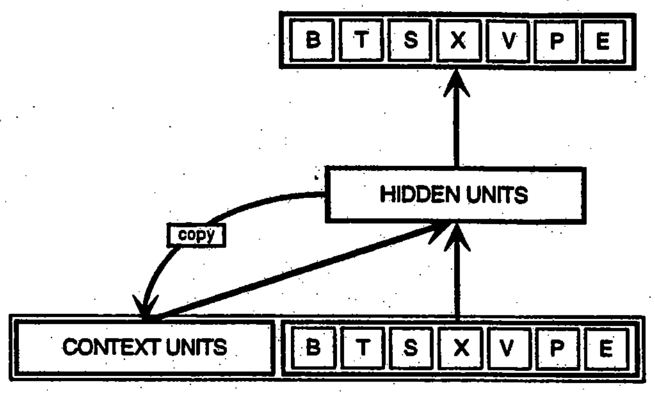

Figure 4. General architecture of the network.

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 *begin* 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.

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 target vector is a seven-bit vector representing element $t + 1$ of the string. The two input vectors are:

- A three-bit vector representing the activation of the hidden units at time $t - 1$, and
- A seven-bit vector representing element t of the string.

2.4. *Training*

On each of 60,000 training trials, a string was generated from the grammar, starting with the 'B.' Successive arcs were then selected randomly from the two possible continuations, with a probability of 0.5. Each letter was then presented sequentially to the network. The activations of the context units were reset to 0 at the beginning of each string. After each letter, the error between the network's prediction and the *actual successor* specified by the string was computed and back-propagated. The 60,000 randomly generated strings ranged from 3 to 30 letters (mean: 7, sd: 3.3)[1].

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.'

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 7a, when the letter 'X' immediately follows 'T,' the network predicts again 'S' and 'X' appropriately. However, as Figure 7b shows, when a second 'X' follows, the prediction changes radically as the network now expects 'T' or 'V.' Note that if the network were not provided with a copy of the previous pattern of activation on the hidden layer, it would activate the four possible successors of the letter 'X' in both cases.

```
String     B t x x v v
           B  T  S  P  X  V  E
Output     00 53 00 38 01 02 00
Hidden           01 00 10
Context          00 00 00
           B  T  S  P  X  V  E
Input      10000 00 00 00 00 00
```

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

```
String     b T x x v v
           B  T  S  P  X  V  E
Output     00 01 39 00 56 00 00
Hidden           84 00 28
Context          01 00 10
           B  T  S  P  X  V  E
Input      00100 00 00 00 00 00
```

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.

```
String    b t X x v v
          B  T  S  P  X  V  E
Output    00 04 44 00 37 07 00
Hidden          74 00 93
Context         84 00 28
          B  T  S  P  X  V  E
Input     00 00 00 00 100 00 00
```

a

```
String    b t x X v v
          B  T  S  P  X  V  E
Output    00 50 01 01 00 55 00
Hidden          06 09 99
Context         74 00 93
          B  T  S  P  X  V  E
Input     00 00 00 00 100 00 00
```

b

Figure 7. a) State of the network after presentation of the first 'X.' b) 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.

We also verified that the network did not accept ungrammatical strings. 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.' During this test, the network is first presented with the 'B' and one of the five letters or 'E' is then selected at random as a successor. If that letter is predicted by the network as a legal successor (i.e., activation is above 0.3 for the corresponding unit), it is then presented to the input layer on the next time step, and another letter is drawn at random as its successor. This procedure is repeated as long as each letter is predicted as a legal successor until 'E' is selected as the next letter. The procedure is interrupted as soon as the actual successor generated by the random procedure is *not* prediced by the network, and the string of letters is then considered 'rejected.' As in the previous test, the string is considered 'accepted' if all its letters have been predicted as possible continuations up to 'E.' Of the 130,000 strings, 0.2% (260) happened to be grammatical, and 99.7% were non-grammatical. The network performed flawlessly, accepting all the grammatical strings and rejecting all the others. In other words, for all non-grammatical strings, when the first non-grammatical letter was presented to the network its activation on the output layer at the previous step was less than 0.3 (i.e., it was *not* predicted as a successor of the previous—grammatically acceptable—letter).

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

'BTSSSSSSSSSSSSSSSSSSSSSSSXXVPXVPXVPXVPXVPXVPXVPXVPXVPXVPXTT TTTTTTTTTTTTTTTTTTTTTTTTTTTTTVPXVPXVPXVPXVPXVPXVPS'

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 more hidden units to reach this performance criterion with much less training. For example, 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. Also, the selection of a threshold of 0.3 is not completely arbitrary. The activation of output units is related to the frequency with which a particular letter appears as the successor of a given sequence. In the training set used here, this probability is 0.5. The activation of a legal successor would then be expected to be 0.5.[2] However, because of the use of a momentum term in the back propagation learning procedure, the activation of correct output units following training was occasionally below 0.5—sometimes as low as 0.3.

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.[3] The graphical result of this analysis is presented in Figure 8A. 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. This grouping by similar predictions is apparent in Figure 8b, which represents an enlargement of the bottom cluster (cluster 5) of Figure 8a. One can see that this cluster groups patterns that result in the activation of the "End" unit: All the strings corresponding to these patterns end in 'V' or 'S' and lead to node #5 of the grammar, out of which "End" is the only possible successor. 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 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.

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, an examination of Figure 8b reveals that patterns ending by 'VV,' 'PS' and 'SXS' endings have been grouped separately by the analysis: they are more similar to each other than to the abstract prototypical pattern that would characterize the corresponding "node."[4] 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 9. However, since the cluster analysis shows that slightly different patterns are produced by the substrings 'BT,' 'BTS' and 'BTSS,' Figure 10 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 backpropagation 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 11 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

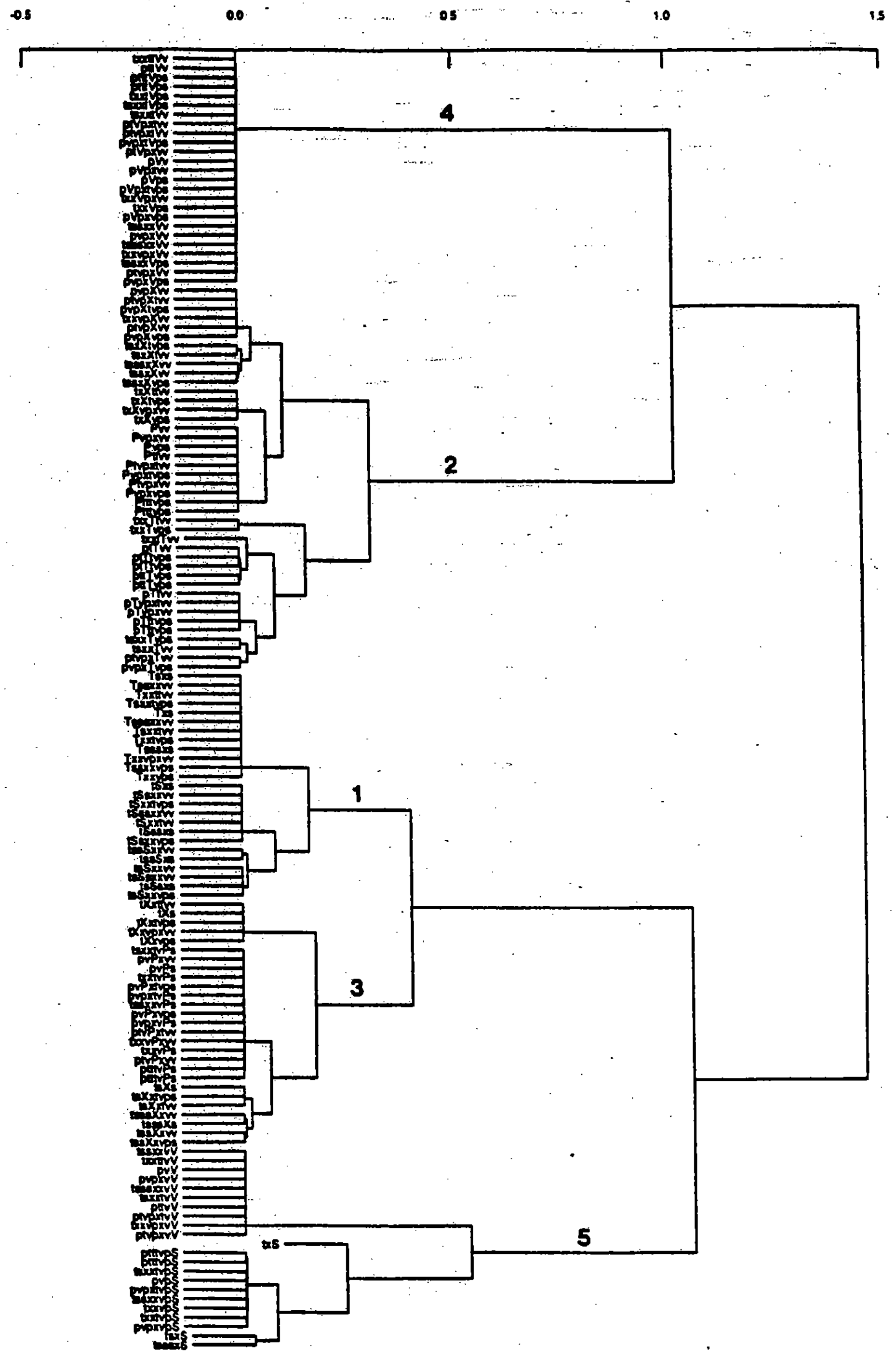

Figure 8A. Hierarchical Cluster Analysis of the H.U. activation patterns after 200,000 presentations from strings generated at random according to the Reber grammar (Three hidden units).

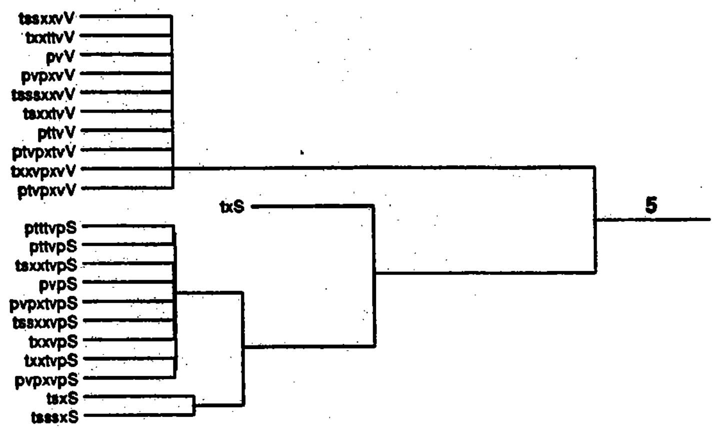

Figure 8B. An enlarged portion of Figure 8A, representing the bottom cluster (cluster 5). The proportions of the original figure have not necessarily been respected in this enlargement.

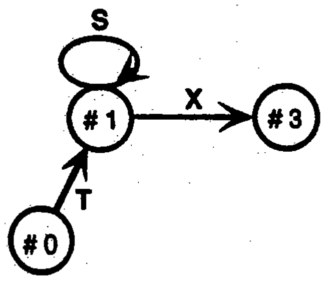

Figure 9. A transition network corresponding to the upper-left part of Reber's finite-state grammar.

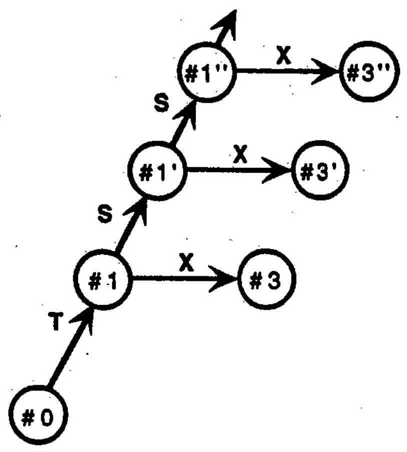

Figure 10. A transition network illustrating the network's true behavior.

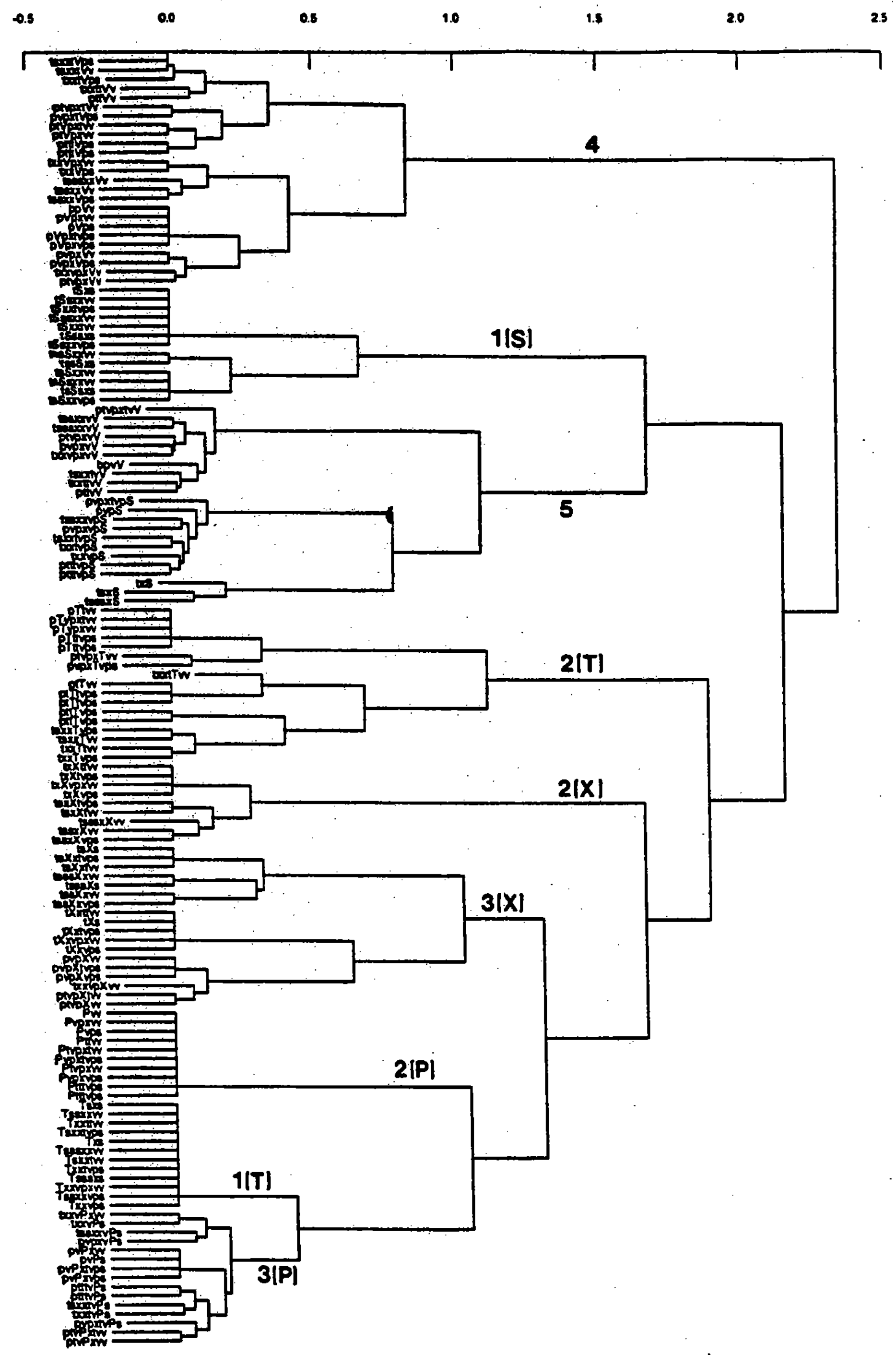

Figure 11. Hierarchical Cluster Analysis of the H.U. activation patterns after 200,000 presentations from strings generated at random according to the Reber grammar (Fifteen hidden units).

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.

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.

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 12 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.

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

Figure 12. The 21 grammatical strings of length 3 to 8.

- 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. For example, following the sequence 'TXX' both 'T' and 'V' are legal successors. However, following the sequence 'TXXVPX,' 'X' is the sixth letter and only 'V' would be a legal successor. This information must therefore also be somehow represented during processing.
- 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 21 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 13a shows that the network expects a 'T' or a 'V' after a first presentation of the second 'X' in the grammar.

Contrast these predictions with those illustrated in Figure 13b, 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.'

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 that 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.

```
String     b t x X v p x v v
           B  T  S  P  X  V  E
Output     00 49 00 00 00 50 00
Hidden     27 89 02 16 99 43 01 06 04 18 99 81 95 18 01
Context    01 18 00 41 95 01 60 59 05 06 84 99 19 05 00
           B  T  S  P  X  V  E
Input      00 00 00 00 100 00 00
```

a

```
String     b t x x v p X v v
           B  T  S  P  X  V  E
Output     00 03 00 00 00 95 00
Hidden     00 85 03 85 31 00 72 19 31 03 93 99 61 05 00
Context    01 07 05 90 93 04 00 10 71 40 99 16 90 05 82
           B  T  S  P  X  V  E
Input      00 00 00 00 100 00 00
```

b

Figure 13. a) State of the network after presentation of the second 'X.' b) State of the network after a second presentation of the second 'X.'

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 (Figures 14a and 14b).

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.

```
String     b t s s x x V v
           B  T  S  P  X  V  E
Output     00 00 00 54 00 48 00
Hidden     44 98 30 84 99 82 00 47 00 09 41 98 13 02 00
Context    89 90 01 01 99 70 01 03 02 10 99 95 85 21 00
           B  T  S  P  X  V  E
Input      00 00 00 00 00100 00
```

a

```
String     b t s s s x x V v
           B  T  S  P  X  V  E
Output     00 00 00 02 00 97 00
Hidden     56 99 48 93 99 85 00 22 00 10 77 97 30 03 00
Context    54 67 01 04 99 59 07 09 01 06 98 97 72 16 00
           B  T  S  P  X  V  E
Input      00 00 00 00 00100 00
```

b

Figure 14. Two presentations of the first 'V,' with slightly different paths.

In order to understand better the kind of representations that encode sequential context 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.

The resulting analysis is shown in Figure 15. We labeled the arcs according to the letter being presented (the 'current letter') and its position in the Reber grammar. 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 'B' 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

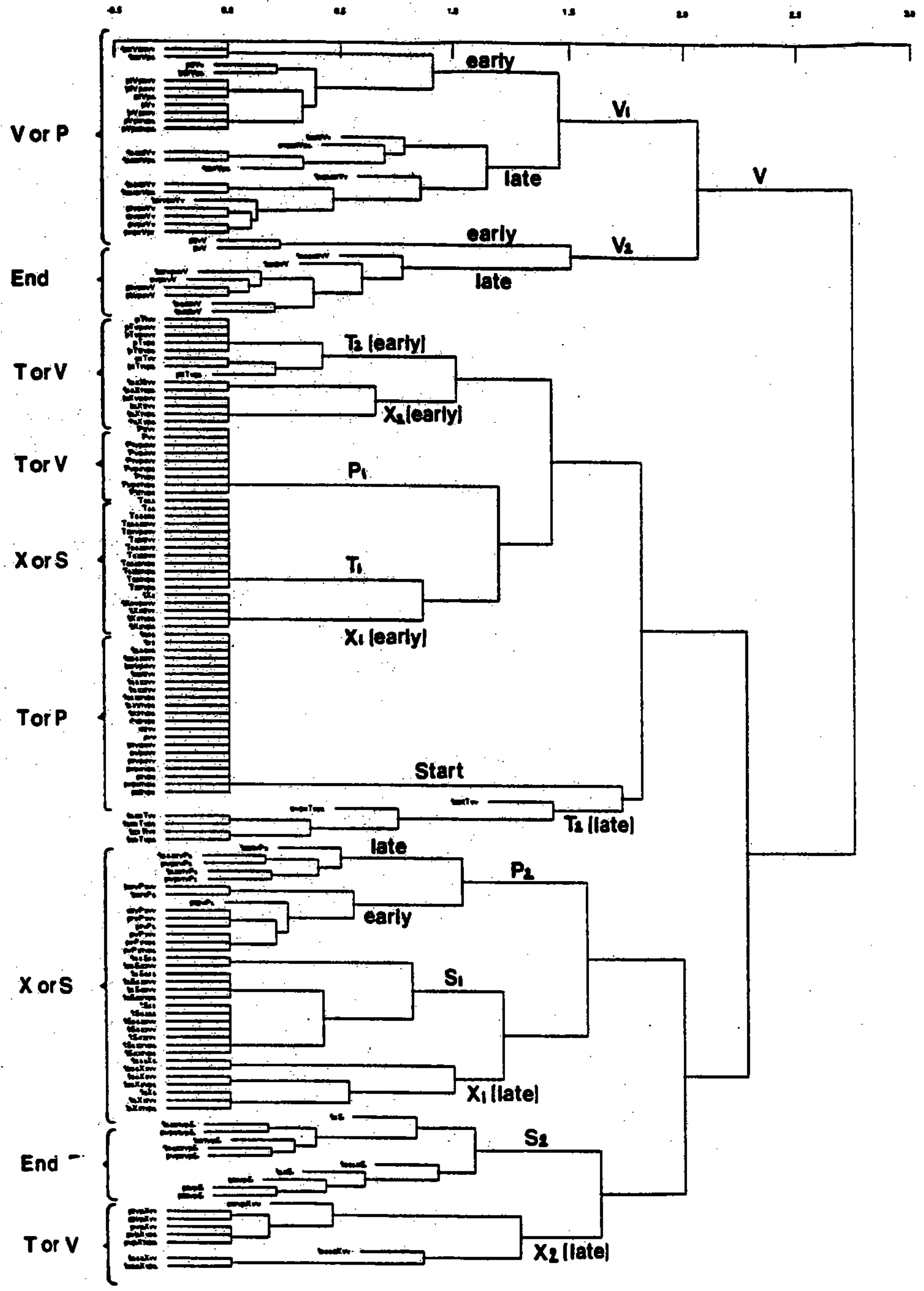

Figure 15. Hierarchical cluster analysis of the H.U. activation patters after 2000 epochs of training on the set of 21 strings.

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. . .'). 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 ouptut, 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 or *shading* on the hidden unit pattern (see Pollack, 1989, for a further discussion of error-driven and recurrence-driven influences on the development of hidden unit patterns in recurrent networks). 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 failed to predict one of two continuations consistent with the grammar and length constraints in 10 other cases.

3.4. Finite state automata and graded state machines

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 not 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 to respond adequately to non-local constraints. Recursive or 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—e.g., 0, .25, .75, 1—can support more than one billion different patterns (4^{15} = 107,374,1824).
- 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.

Because of its inherent ability to use *graded* rather than finite states, the SRN is definitely not a finite state machine of the usual kind. As we mentioned above, we have come to consider it as an exemplar of a new class of automata that we call *Graded State Machines.*

In the next section, we examine how the SRN comes to develop appropriate internal representations of the temporal context.

4. Learning

We have seen that the SRN develops and learns to use compact and effective representations of the sequences presented. These representations are sufficient to disambiguate identical cues in the presence of context, to code for length constraints and to react appropriately to atypical cases.[5] How are these representations discovered?

As we noted earlier, in an SRN, the hidden layer is presented with information about the current letter, but also—on the context layer—with an encoding of the relevant features of the previous letter. Thus, a given hidden layer pattern can come to encode information about the relevant features of two consecutive letters. When this pattern is fed back on the context layer, the new pattern of activation over the hidden units can come to encode information about three consecutive letters, and so on. In this manner, the context layer patterns can allow the network to maintain prediction-relevant features of an entire sequence.

As discussed elsewhere in more detail (Servan-Schreiber, Cleeremans & McClelland, 1988, 1989), learning progresses through three qualitatively different phases. During a first phase, the network tends to ignore the context information. 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 its possible successors. For example, after only 100 epochs of training, the response pattern generated by 'S_1' and the corresponding output are almost identical to the pattern generated by 'S_2', as Figures 16a and 16b demonstrate. At the end of this phase, the network thus predicts all the successors of each letter in the grammar, independently of the *arc* to which each letter corresponds.

```
Epoch     100
String     b S s x x v p s
           B  T  S  P  X  V  E
Output    00 00 36 00 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     00 00 100 00 00 00 00
```

a

```
Epoch     100
String     b s s x x v p S
           B  T  S  P  X  V  E
Output    00 00 37 00 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     00 00 100 00 00 00 00
```

b

Figure 16. a) Hidden layer and output generated by the presentation of the *first* S in a sequence after 100 epochs of training. b) Hidden layer and output patterns generated by the presentation of the *second* S in a sequence after 100 epochs of training.