Report Number: WUCS-93-52

1993

Tail-Recursive Distributed Representations and Simple Recurrent Networks

Stan C. Kwasny and Barry L. Kalman

Representation poses important challenges to connectionism. The ability to structurally compose representaitons is critical in achieving the capability considered necessary for cognition. We are investigating distributed patterns that represent structure as part of a larger effort to develop a natural language processor. Recursive Auto-Associative Memory (RAAM) representations show unusual promise as a general vehicle for representing classical symbolic structures in a way that supports compositionality. However, RAAMs are limited to representations for fixed-valence structures and can often be difficult to train. We provide a technique for mapping any ordered collection (forest) of hierarchical structures (trees) into a set of... Read complete abstract on page 2.

Tail-Recursive Distributed Representations and Simple Recurrent Networks

Stan C. Kwasny and Barry L. Kalman

Complete Abstract:

Representation poses important challenges to connectionism. The ability to structurally compose representaitons is critical in achieving the capability considered necessary for cognition. We are investigating distributed patterns that represent structure as part of a larger effort to develop a natural language processor. Recursive Auto-Associative Memory (RAAM) representations show unusual promise as a general vehicle for representing classical symbolic structures in a way that supports compositionality. However, RAAMs are limited to representations for fixed-valence structures and can often be difficult to train. We provide a technique for mapping any ordered collection (forest) of hierarchical structures (trees) into a set of training patterns which can be used effectivelyin training a simple recurrent network (SRN) to develop RAAM-style distributed representations. The advantages in our technique are three-fold: first, the fixed-valence restriction on structures represented by patterns trained with RAAMs is removed; second, representations resulting from training corresponds to ordered forests of labeled trees thereby extending what can be represented in this fashion; and third, training can be accomplished with an auto-associative SRN, making training a much more straightforward process and one which optimally utilizes the n-dimensional space of patterns.

Tail-Recursive Distributed Representations and Simple Recurrent Networks

Stan C. Kwasny
Barry L. Kalman

WUCS-93-52

November, 1993

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, Missouri 63130-4899

sck@cs.wustl.edu
barry@cs.wustl.edu

This material is based upon work supported by the National Science Foundation under Grant No. IRI-9201987.

Tail-Recursive Distributed Representations and Simple Recurrent Networks

Stan C. Kwasny
Barry L. Kalman

Department of Computer Science
Washington University, Campus Box 1045
St. Louis, Missouri 63130
U.S.A.
sck@cs.wustl.edu
barry@cs.wustl.edu
(314) 935-6123

ABSTRACT

Representation poses important challenges to connectionism. The ability to structurally compose representations is critical in achieving the capability considered necessary for cognition. We are investigating distributed patterns that represent structure as part of a larger effort to develop a natural language processor. Recursive Auto-Associative Memory (RAAM) representations show unusual promise as a general vehicle for representing classical symbolic structures in a way that supports compositionality. However, RAAMs are limited to representations for fixed-valence structures and can often be difficult to train.

We provide a technique for mapping any ordered collection (forest) of hierarchical structures (trees) into a set of training patterns which can be used effectively in training a simple recurrent network (SRN) to develop RAAM-style distributed representations. The advantages in our technique are three-fold: first, the fixed-valence restriction on structures represented by patterns trained with RAAMs is removed; second, representations resulting from training corresponds to ordered forests of labeled trees thereby extending what can be represented in this fashion; and third, training can be accomplished with an auto-associative SRN, making training a much more straightforward process and one which optimally utilizes the n-dimensional space of patterns.

1. Introduction

Natural language processing (NLP) relies on the ability to represent hierarchic (nested) structures in formulating interpretations. Processing requirements are reflected in the choice and arrangement of constituents in these structures. Traditionally, structure evolves during sentence processing compositionally. Smaller symbolic structures are composed into larger ones along the chosen processing path. Each structure occupies a variable amount of storage and can be decomposed systematically into its smaller parts.

Structures which require unbounded storage introduce a variety of problems for connectionism. Connectionist models have considerable difficulty learning, representing, and manipulating such structures within fixed-length vectors with limited computational resources and with finite precision numeric calculations. To achieve the compositional ability necessary to support connectionist NLP, structural representations must either be supported through symbolic means in a hybrid arrangement, or fixed-sized distributed representations must exist which are capable of representing a wide range of structures and can be manipulated holistically during processing.

Recursive Auto-Associative Memory (RAAM) is a connectionist network architecture created by Jordan Pollack (1989; 1990) that enables simple structures to be represented as distributed patterns. These patterns can be composed from structures and decomposed into constituent parts just as symbolic hierarchical structures can. Plate (1991) has compared convolution-correlation (holographic) memories to RAAMs and concluded there are advantages and disadvantages to each, making neither clearly superior.

This paper presents some new developments to RAAMs which make them more general and easier to train. We are investigating the use of RAAMs in support of natural language processing, but any type of representation that requires embedded structure could utilize RAAMs. Representation is a critically important issue for connectionism, particularly in view of attacks on the inherent limitations of connectionism and the requirement that the success of connectionism be determined by comparison with traditional approaches. Localist representation schemes, especially with respect to language processing, have their limitations, as discussed by Elman (1991), and distributed representations have yet to evolve to their fullest potential.

Pollack's RAAMs provide distributed activation patterns over a fixed-sized set of units that approximate symbolic hierarchical (tree-like) structures of fixed valence[1] as an emergent property of the network. During training, specific examples of symbolic structures are presented, and subnetworks for composition and extraction co-evolve as the network learns how to simultaneously compose and decompose structures. Limited generalization to structures not used in training has been shown to occur.

[1] Valence refers to the number of subtrees, called children, associated with each node in a tree. A tree of fixed valence is a k-ary tree in which each non-terminal node has a fixed k number of children. Presumably, some of these children could be empty subtrees, effectively making the fixed valence, k, an upper bound on the number of children at each node.

1.1. Previous work on RAAMs

RAAM networks have been the focus of several studies. In his original studies, Pollack (1989; 1990), demonstrated feasibility as well as limited generalization capabilities in experiments with representing letter sequences and experiments with a small collection of syntactic parse trees. In testing the generative capacity of the networks, some generalization was shown to occur.

Berg (1992) developed RAAM-style representations in compressing the constituent structure of sentences. He showed that, in principle, recursive sentence structure could be represented without bound and that the method could also achieve some lexical disambiguation.

Chalmers (1990) demonstrated that distributed RAAM patterns could be fruitfully operated upon holistically. With distributed patterns as inputs, he trained a network to extract various pieces of the structure implicit in those patterns and experimented with networks that could meaningfully map one RAAM pattern to another in a holistic manner.

Chrisman (1991) explored this capability further. Rather than following Chalmers, who used a previously trained RAAM network to generate distributed patterns for the holistic mappings (which Chrisman calls transformational inference), Chrisman included the transformational mapping as part of training (which Chrisman calls confluent inference). The technique is based on an architecture called the dual-ported RAAM and is illustrated with examples of English-Spanish associations. Structures used in these examples, however, are still fixed valence trees.

In an excellent series of studies, Blank, et al (1992) further investigated holistic properties of the distributed RAAM patterns and the use of a sequential RAAM (SRAAM) (which we describe in the next section) to process sequences of words as sentence forms. A crude form of lexical representation emerges when distributed patterns involving a particular term in a variety of contexts are averaged. These averages are shown to cluster according to selected semantic properties.

Sperduti (1993a; 1993b) has shown that a RAAM-like architecture, called an LRAAM, can be utilized to represent a graph structure whose nodes are labeled and which may contain cycles. No generalization abilities are claimed.

1.2. RAAM Limitations

RAAMs in their current form have several limitations. Earlier, we mentioned that structures must be of fixed valence in order to be represented under the method. Because of this, the connectivity between nodes and their constituents in the structures to be represented is directly reflected in the connectivity between layers in the RAAM network. That is, the network resembles the structure in its connectivity and each node has a uniform number of children according to the valence of the structures.

Furthermore, RAAM training is arguably one of the most difficult training tasks for neural networks. Chrisman uses a complex series of training steps each modifying the weights according to a training regimen. Others stage training to develop new representations from

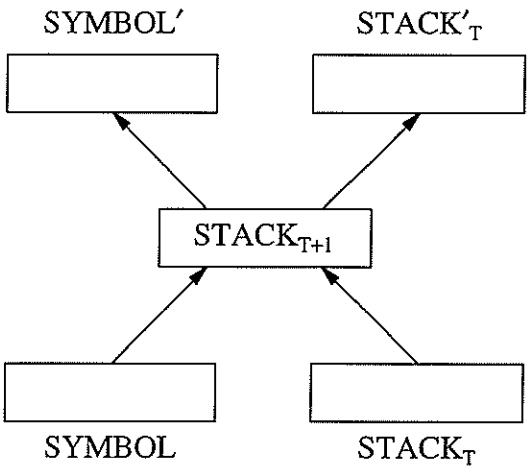


Figure 1. Sequential RAAM Architecture for representing a stack. The input and output layers are each separated into a symbol pattern and a distributed STACK pattern. The subscript, $T \geq 0$, is used to indicate how many symbols are present in the stack. The hidden layer pattern, $STACK_{T+1}$, represents a stack containing the additional symbol.

previous (shallower) ones, but deeper training examples depend on the nuances of shallower ones in arbitrary ways and there is no obvious way to assure that an optimal representation scheme has been found for all structures in the representation space. Berg conducts training by "unrolling" the structure to the deepest level and training the network by making virtual copies of a single network. Reported training times are from one to three cpu months for the simple set of sentences tested.

Finally, while symbolic labels are usually found in the internal nodes of most symbolic structures, the original RAAM architecture did not allow for them. LRAAMs are a variation that permits labels, but at the expense of generalization.

Progress is being made independently on all of these issues, according to Pollack (personal communication). This paper examines a sequential RAAM variant, capable of developing distributed representations for ordered collections of labeled structures (forests of trees) of arbitrary valence. The representations are optimized in training over the entire n-dimensional representation space. The SRAAM, is discussed by both Pollack (1990) and Blank et al. (1992). Under this scheme, training is still auto-associative, but reduces to little more than that required for a simple recurrent network (SRN) as described by Elman (1990).

We argue that SRAAMs are preferable to RAAMS in a number of ways and present an effective method for training them along with a demonstration using collections of structures of various depths and valences. Performance is based on the ability of the SRAAM network to encode and decode structures effectively for increasingly larger structures. We show that SRAAMs generalize remarkably well.

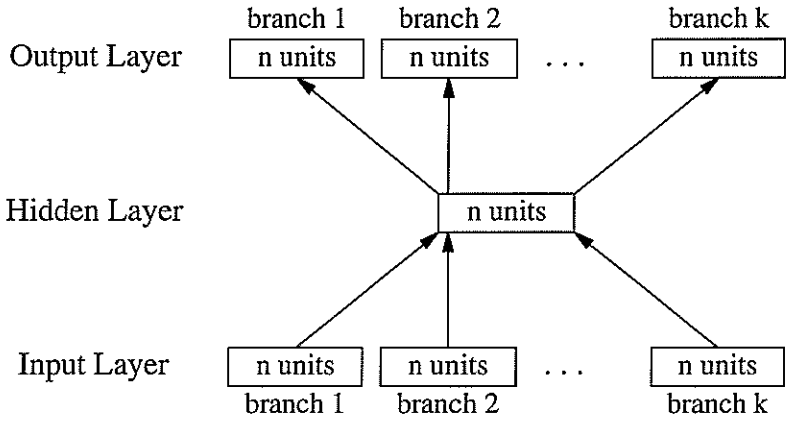


Figure 2. General RAAM Architecture for k-ary tree structures. Each distributed pattern must contain n units and each internal tree node must contain k subtrees, although some of these may be empty.

2. SRAAM Architecture

First, we introduce the SRAAM architecture. For illustration purposes, consider the representation of a simple stack of symbols. As Figure 1 shows, the input and output layers are identically divided into the symbol part and a distributed pattern part which represents the stack. The hidden layer is smaller than the input and output layers and contains the same number of units as the distributed stack pattern. Training is auto-associative which means that the targets for the output layer are taken to be exactly the input patterns presented. From this architecture, each hidden layer pattern effectively represents the stack resulting from a push of the symbol onto the stack represented in the input layer. Any stack representation, with the exception of the empty stack representation, can be popped by application of the weights connecting the hidden layer to the output layer. Applying these weights to a stack yields a symbol and a (one symbol smaller) stack.

Generalizing from the SRAAM leads to the general RAAM architecture as shown in Figure 2. Again, training is auto-associative, but patterns that develop at the hidden layer now represent a compression of the k branches that have been presented. Each branch may either be the chosen representation of a (terminal) symbol or the distributed representation of a smaller structure. In this way, hierarchical structures of increasing depth, but maximum valence k, can be represented.

After training, the network can be utilized to build hierarchical (tree) structures recursively. By examining the network one weight-layer at a time, this functionality can be summarized. The encoding part of the network is represented by the weights connecting input layer to hidden layer. When these weights are applied, a compressed representation results at the hidden layer which when acted upon by the weights connecting hidden to output layer

produces a close approximation to the k input branches once more. Thus the network learns to approximate an identity mapping of inputs to outputs. Each branch may also be a previously-trained hidden-layer activation pattern that represents the compression of a smaller substructure. In this manner, the composition of nested structures is supported.

To recover the entire structure, the decoding (reconstructor) circuitry must be applied iteratively and so training must produce a network that performs to a relatively tight tolerance. The auto-associative nature of training says that the targets are identical to the inputs, but since training is never perfect, there will always be some error present in the output patterns. When structures are encoded into distributed patterns, the information is not perfectly represented and, when decoded, will never perfectly reconstruct the structure. If encoding is performed iteratively, there can be an accumulated error effect that disturbs the decoding circuitry's ability to recover the structure. This means that very tight training must be performed, particularly with respect to the distributed pattern outputs. Stolcke (1992) noticed that it was possible for errors to interfere to the point where terminal nodes of the tree structure were mistaken for non-terminals and further decoding resulted in garbage. His solution was to add a unit to the pattern which is an indicator showing whether it was a terminal representation.

3. Symbolic Mappings

We wish to represent compositions of nested structures. We define the nature of these structures, in the most general terms, as forests of labeled trees. In our approach, we process the structure symbolically to produce a sequence of symbols so that training may proceed sequentially. In this section, we formalize our terms and provide a sketch of the algorithm for accomplishing this. Appendix A contains more description and an implementation in Scheme.

A *tree* can be defined as a collection of one or more labeled nodes that can be partitioned into a root node and a disjoint ordered set of zero or more trees called subtrees of the root. A *forest* is an ordered set of zero or more trees. The ordered collection of subtrees under any root form a forest. A binary tree is a finite set of labeled nodes which is either empty or has a root node and two disjoint binary subtrees, called the left subtree and the right subtree.

Any forest of trees $\{T_1, T_2, \ldots, T_n\}$ can be mapped into a single binary tree, B and vice versa. This is accomplished by mapping the root of T_1 to the root of B, mapping the forest of subtrees under the root of T_1 to the left subtree of B, and mapping the remaining forest $\{T_2, \ldots, T_n\}$ to the right subtree of B. Note that this mapping is reversible by reversing each part of the mapping. An example of this mapping is shown in Figure 3a.

Any binary tree can be mapped to a sequence of labels in which empty subtrees are explicitly indicated. This is done by traversing the binary tree in preorder. In a preorder traversal, first the root node is processed, then the left subtree is traversed recursively in preorder, and finally the right subtree is traversed recursively in preorder. Processing a labeled node results in outputting the label as part of the sequence and traversing an empty binary tree results in adding an empty symbol (we use a dot "•") to the sequence.

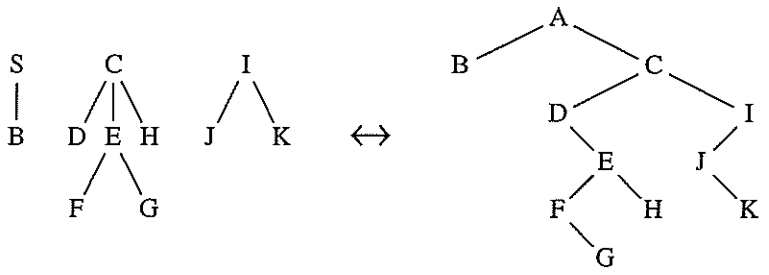
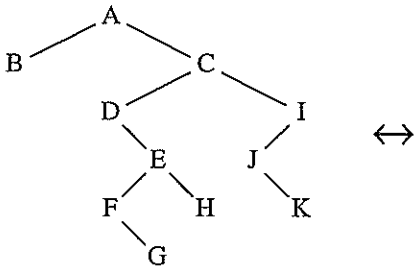


Figure 3a. Transformation from forest of trees to binary tree.



A B • • C D • E F • G • • H • • I J • K • • •

Figure 3b. Transformation from binary tree to list. The dot (•) represents an empty tree.

Note that this mapping is also reversible from an appropriate sequence to a binary tree. Observe that each root node in the sequence is followed first by the sequence for its left subtree and next by the sequence for its right subtree. A dot indicates an empty subtree and observe that for any well-formed binary tree, there is exactly one more dot than non-dot member of the sequence. This means that it is easy to distinguish well-formed subtrees and, therefore, to determine exactly where the left subtree ends and the right subtree begins after any root node label. An example of this mapping is shown in Figure 3b.

In the experiments reported here, we use single-tree forests. In the first experiment, derivation trees are generated from a context-free grammar and divided into training and testing patterns. A second experiment uses structures that arise as intermediate and final results during Natural Language parsing in a connectionist parser under development (see, Kwasny & Faisal, 1992; Kwasny & Kalman, 1992).

4. Recurrent Training

We avoid the fixed-valence problem of the general RAAM architecture by symbolically transforming each structure into a sequential list of symbols prior to presentation to the network. With this change, the network remains as simple as that required for the stack example shown in Figure 1. This network, in fact, is merely a minor variation of an SRN and therefore

can be trained as such.

To train an SRAAM using an SRN training algorithm, we modify the algorithm to work in an auto-associative manner, taking its targets from the input patterns rather than requiring them to be provided for each pattern. Each training sequence starts by presenting the coded first symbol along with the empty pattern, ε (a chosen pattern representing the empty structure — we use a vector of all zeros). Subsequent presentations require that the activation pattern from the hidden layer in the previous step be copied to the input layer as context. During training, copying ceases with the beginning of the next sequence which again uses the ε pattern. Figure 4 shows how training proceeds. On each step, SYMBOL_T is a vector of units activated from the encoding of the next symbol in the sequential list. The DPAT_T portion is initially ε and on each subsequent step in the sequence its value is the pattern present on the hidden layer of the previous step. The output layer is likewise divided into two segments, the symbol part and the distributed pattern part. Values closely approximating the corresponding input values should occur as the network learns the mapping. We use the prime (′) notation to indicate this approximating behavior. Training of the network reinforces its ability to compress the previous distributed RAAM representation and the current symbol. The two weight layers co-evolve during training to work in unison: the decoding part of the network learns to approximate the inverse of the encoding part.

Note that the network is partially supervised (the symbol part) and partially unsupervised (the distributed pattern part). The localist symbol patterns are chosen and become the targets

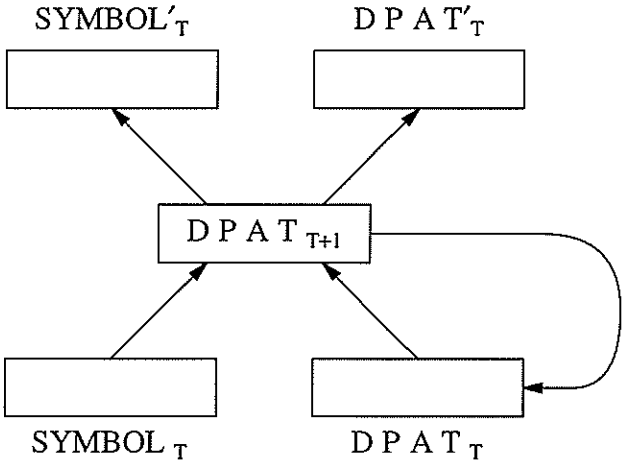


Figure 4. Sequential RAAM Training. In the input layer, symbols are coded as ±1 activations, while the distributed pattern, DPAT, has activations over the interval [-1,1]. The targets for the output layer are identical to these patterns, but since some error exists when the network produces its output activation, we use prime (′) to show that.

for supervised learning that do not change from epoch to epoch. The distributed structure patterns, however, evolve as a by-product of training and are globally optimized during training.

Contrary to RAAM training described in the literature, it becomes unnecessary to first find representations for small structures so that training can combine them into larger structures. All structures reduce to a sequence of symbols and training is much more uniform. The architecture of the network does not reflect the valence or any other feature of the structure itself. Structures are encoded from the sequence left-to-right and decoding produces structures right-to-left[2]. The sequences are processed tail-recursively (iteratively) and therefore the resultant representations are tail-recursive in this sense.

Besides auto-associativity, training an SRAAM differs in another way from training an SRN. From Figure 4, we see that for each unit k associated with the distributed pattern portion of the output, the difference between its target value, $DPAT_{Tk}$, and the output of the network, $DPAT'_{Tk}$, contributes to the error function. Since the activation of $DPAT_{Tk}$ is changing as training proceeds, a correction term must be added to the gradient computation associated with the connections from that part of the input layer, $DPAT_T$, to the hidden layer $DPAT_{T+1}$. In our experience, without this correction term, calculations during training are sufficiently incorrect to adversely affect success.

Most neural network research utilizes an error function based on mean square error (MSE) and measure the performance of their networks in proportion to this calculation. While the main points of this paper are independent of the specific choice of error function, our experiments were conducted using a modified MSE function. For completeness, we will provide some of the details of that function here. See Kalman & Kwasny (1991; 1993) for more details.)

We have demonstrated that a network can be very wrong on a few important unit activations even when the mean square error is very small. This has led us to utilize the Kalman-Kwasny error function which avoids many of these problems. The error function itself can be viewed as a variation on MSE and is calculated as:

$$\Phi = \sum_p \sum_i \frac{(t_{pi} - a_{pi})^2}{1 - a_{pi}^2}$$

where p runs over patterns and i over output units. Target, t_{pi}, and activation, a_{pi}, are specific to an input pattern presentation and an output unit. Calculating the derivative, leads to:

$$\frac{\partial \Phi}{\partial Y} = \sum_p \sum_i \frac{2}{(1 - a_{pi}^2)^2} (t_{pi} - a_{pi})((t_{pi} - a_{pi})\, a_{pi} - (1 - a_{pi}^2)) \frac{\partial a_{pi}}{\partial Y}$$

where Y represents any weight or bias of the network.

[2] Since each structure is a sequence, that is the only important property that must be preserved. Decoding of the structure always reverses the order of encoding in typical LIFO fashion. This leaves open the question of whether it is better to encode left-to-right or right-to-left. We have observed situations in which right-to-left may be better.

For SRAAM networks, the distributed pattern portion varies during training and, therefore, must be considered in calculating the error. As mentioned earlier, the architecture is partially supervised and partially unsupervised. For the unsupervised, distributed pattern units, the error function would be:

$$\Phi = \sum_T \sum_k \frac{(DPAT_{Tk} - DPAT'_{Tk})^2}{1 - DPAT'^2_{Tk}}$$

For ordinary and simple recurrent networks, the term $DPAT_{Tk}$ would be identical from epoch to epoch, but not for RAAM SRNs. Our training system correctly accounts for this variation. From this we get the modified gradient computation:

$$\frac{\partial \Phi_{DPAT}}{\partial Y} = SRN' + \sum_p \sum_i \frac{2(DPAT_{Tk} - DPAT'_{Tk})}{(1 - DPAT'^2_{Tk})} \frac{\partial DPAT_{Tk}}{\partial Y}$$

where SRN′ is the derivative for SRNs as shown above.

Our training algorithm is a modified conjugate gradient training algorithm that utilizes an improved line search method based on an adaptive step size bounding algorithm and Brent's derivative-free line search algorithm (see Kalman, 1990). We have argued (Kalman & Kwasny, 1992) that use of our improved error function in conjunction with a hyperbolic tangent squashing function leads to the best results.

5. Representing Structures as SRAAMs

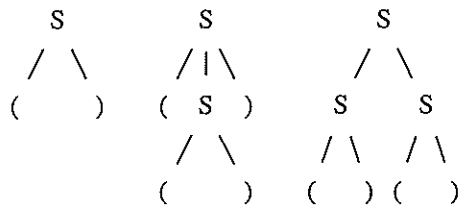
In this section, we provide experimental evidence that these methods work well for structure composition. The first example comes from a context-free grammar that can be used productively to demonstrate generalization over a large number of examples. The second example is derived from a more practical problem of representation related to building parse trees while processing sentences.

5.1. Representing Context-Free Derivation Trees

In order to test our approach to representing structures, we used a simple context-free grammar to generate derivation tree structures. The grammar consists of three rules:

$$\mathbf{S} \rightarrow (\ )$$
$$\mathbf{S} \rightarrow (\ \mathbf{S}\ )$$
$$\mathbf{S} \rightarrow \mathbf{S}\ \mathbf{S}$$

These rules ambiguously describe all strings of balanced parentheses. Each left parenthesis has a matching right parenthesis and are properly nested. We define the level, n, of a derivation tree for this grammar as the maximum number of non-terminal, **S**, symbols encountered along any path from root to terminal symbol. With this definition, the trees:



are level one, two, and two respectively. There is only one possible level-one tree, three trees of level two or less (the ones shown above), 13 trees of level three or less, 183 trees of level four or less, 33,673 trees of level five or less, etc. In general, the count of derivation trees by level, C(n), is given by the recurrence formula:

$$\begin{cases} C(0) = 0 \\ C(1) = 1 \\ C(n) = C^2(n-1) + 2\,C(n-1) - C(n-2) - C^2(n-2) \end{cases}$$

We generated 30 sample derivation trees by randomly applying one of the three rules starting at a root node, **S**. We level-limited the trees generated so that they were all level four or less by forcing the selection of the terminal rule whenever generating from an **S** at level three. These trees were symbolically mapped to sequences containing the four symbols, { s, (,), • } using the method described in section 3. This resulted in the three sequences:

s (•) • • •
s (• s (•) • •) • • •
s s (•) • • s (•) • • • •

A context-free grammar that describes the language of these sequences consists of the four rules:

$$\begin{aligned} \mathbf{S} &\rightarrow \mathbf{X}\bullet \\ \mathbf{X} &\rightarrow s\,\mathbf{X}\,\mathbf{S} \\ \mathbf{X} &\rightarrow s(\bullet\mathbf{X})\bullet\bullet \\ \mathbf{X} &\rightarrow s(\bullet)\bullet\bullet \end{aligned}$$

where { **S**, **X** } is the set of non-terminals and { s, (,), • } is the set of terminals. The 30 sequences generated 930 individual patterns which were used to train an SRAAM network whose symbol portion of the input layer contained 4 units, one for each symbol, and whose distributed patterns contained 10 units.

Training required 450 derivative epochs and 4,054 functional epochs[3] which took 6,900 seconds (just under 2 hours) cpu time on a SPARC 10/41 machine. Upon convergence,

[3] The conjugate gradient method makes it difficult to report epochs in a way comparable to a method like back-prop. A derivative epoch consists of presenting all patterns, calculating the error and the gradient, and adjusting weights accordingly. It runs in time proportional to the fourth power of the number of hidden units H^4. A functional epoch consists of presenting all patterns and calculating the output pattern of the network. It runs in time proportional to the square of the number of hidden units H^2.

performance was perfect on the 930 training patterns. To determine generalization capabilities, we next generated all 183 derivation trees of level limit 4 and symbolically mapped these to sequences which generated 7,787 patterns. The performance was again perfect for all patterns. Finally, we randomly generated a collection of 1,000 unique derivation trees whose level was 5 (one greater than the maximum level used in the training patterns) and tested the network with the 57,328 patterns that resulted. The network generated the symbolic output in every case and was within 0.054 of the target activation value, as determined by the infinity norm, for every unit in the distributed pattern output. Apparently the network had learned the grammar of the sequences very well. Section 6 will explore the nature of the representations derived in this example.

5.2. Representing Syntactic Parse Trees

In a second experiment, we derived 25 different syntactic parse tree forms that a particular grammar was capable of producing on steps leading to a valid parse. These are shown in Figure 5. Taken as single tree forests, these structures served as training examples for an SRAAM. Training was successful using 16 units for the representation and 17 units for the symbols (one unit for each symbol). Following training, we tested the performance of the iterated encoding and decoding process by testing with these same structures. While training and testing with identical data is not a valid evaluation of the training, in this case it does show the effectiveness of the encoding/decoding subnetworks when iteratively applied. Observe that training only attempts to perfect the individual single-step operations. The real effectiveness of the method lies in its ability to repeatedly and consistently operate on the representation.

In the course of developing representations for the 25 unique sequences of symbols, a total of 75 unique RAAM patterns were developed counting the empty pattern. These included representations for all sequences of lengths 0 through 15. Training produced only one small structural error in one of the 25 sentences, which led to redundancy in one part of the sequence. There also were 4 symbolic errors out of a possible 222 symbols tested for decoding, but these were uniformly minor, differing in one coding position.

Given the nature of our data, there is a high degree of similarity in the initial symbols of many of the sequences. In fact, every sequence begins with an "s". To prevent overemphasis of presentations coming from these common prefixes of sequences, we eliminated all but unique presentations from the error calculation. This brings training into better balance and permits errors occurring later in the sequence to get full attention.

6. Cluster Analysis

Hierarchical clustering analysis was used to determine general strategies on which representations were based. The clustering technique examines a collection of distributed representations (vectors) and compares them pair-wise by computing the Euclidean distance between each pair. This distance provides a one-dimensional measure of closeness and is the

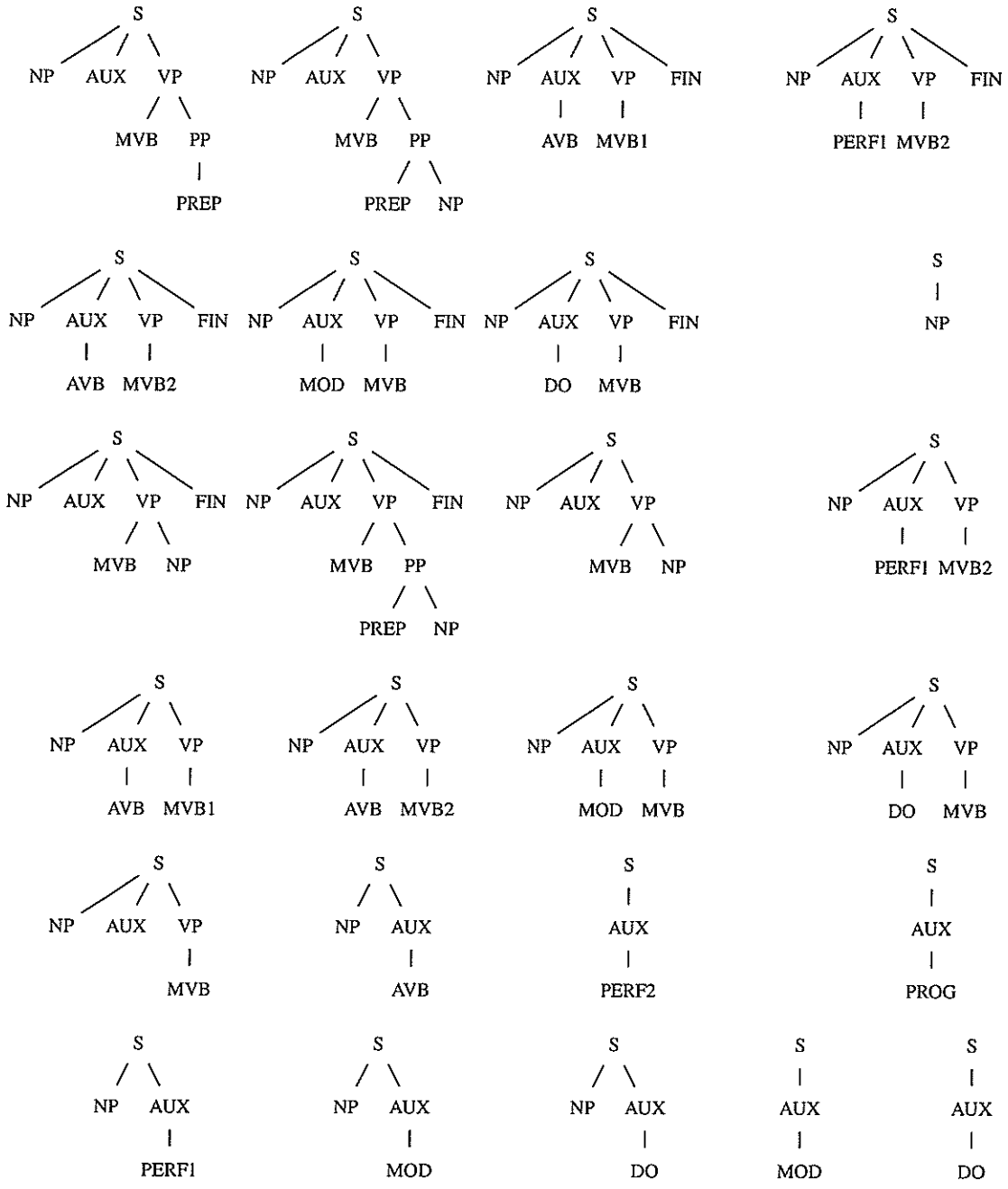


Figure 5. The 25 tree structures used in these experiments. The empty tree is not shown.

basis for building a binary cluster tree. Starting with the set of all vectors, the clustering algorithm finds the closest two vectors, groups them, and computes an average vector of the group. The two selected vectors are replaced in the set with their average and the process is repeated until only one vector remains which is the average of the whole collection. This process results in a binary tree. To display the binary tree, the left and right subtrees under each node are arbitrarily ordered to give only one of many possible orderings of the vector patterns.

Along the vertical axis, the patterns are enumerated and along the horizontal axis, the distance between two vectors or averages is shown in scale.

In order to determine the basis for the representation, we developed cluster trees for each of the two experiments. From this we are able to give a better explanation and interpretation of each network's overall organization.

6.1. Clustering the parenthesis language representations

For the context-free parenthesis balancing language, it was apparent that good generalization had occurred. We further analyzed the 930 distributed patterns that the hidden unit activations had produced over the original 30 training sequences to understand these results more fully. The resulting cluster tree is shown in Figure 6.

Clustering shows that there are particular groupings within the 10-dimensional space which are quite closely clustered. We arbitrarily identified 12 groupings and numbered these 1-12 as shown in the figure. Since every training sequence begins with the symbol s initially presented to the network along with the empty pattern, this yields an identical activation pattern across the hidden layer for all 30 sequences. This can be identified as cluster 10 in the figure. From here, each of the sequences follows a different trajectory through the 12 clusters. From these observations, we are able to derive the 12-state finite state machine (FSM) which is shown in Figure 7 (a). This FSM accepts all 30 sequences.

Notice that the clustering diagram provides a principled way of grouping points in 10-dimensional space. If we start by grouping every point into a one-state FSM, this machine accepts any string over the four-symbol alphabet. By grouping the points into the two major groupings given by the left and right subtrees of the binary cluster tree, we get a two-state FSM with an "s" state, corresponding to states 8–12, and a second state for transitions on { (,), • }, as shown in Figure 7 (b). As we progress left-to-right in the cluster tree, the distances between clusters, as indicated by the lengths of horizontal segments, provide a metric for determining which clusters are furthest apart and hence which state to split to form an FSM with more states than the previous. The three-state FSM is shown in Figure 7 (c). At the four-state machine, as seen in Figure 7 (d), each state can be associated with a single symbol in the alphabet { s, (,), • } according to what symbol led to being in that state. The five-state FSM is shown in Figure 7 (e). This process can be continued up to the extreme in which every unique data point is in a separate state. Such an FSM would accept only those sequences represented in the training data and is therefore uninteresting.

The ordered collection of FSMs show how the space is organized. SRAAM training co-evolves two separate, but closely related networks, one for encoding and one for decoding. Here, the FSM shown is an encoding machine which constructs a representation for a given sequence. The decoding process is simply the machine run in reverse. That is, each state of the FSM corresponds to a cluster of distributed patterns such that when the decoding circuitry acts on one of those patterns, it must be able to retrieve the symbol just encoded. Obviously, in the FSMs we are seeing a sensible way for the network to organize the space to accomplish

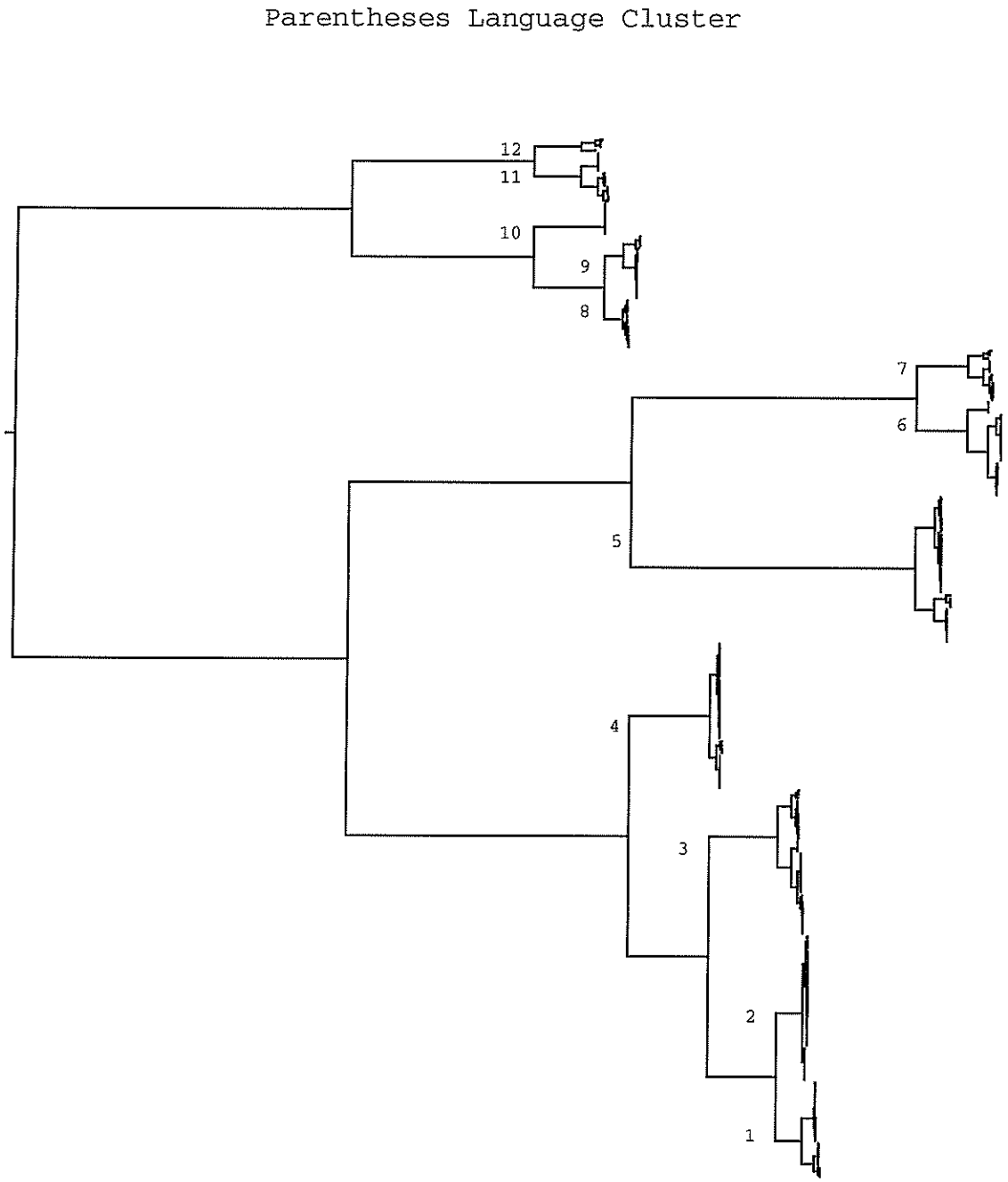


Figure 6. Cluster Analysis of Parenthesis Language Sequences.

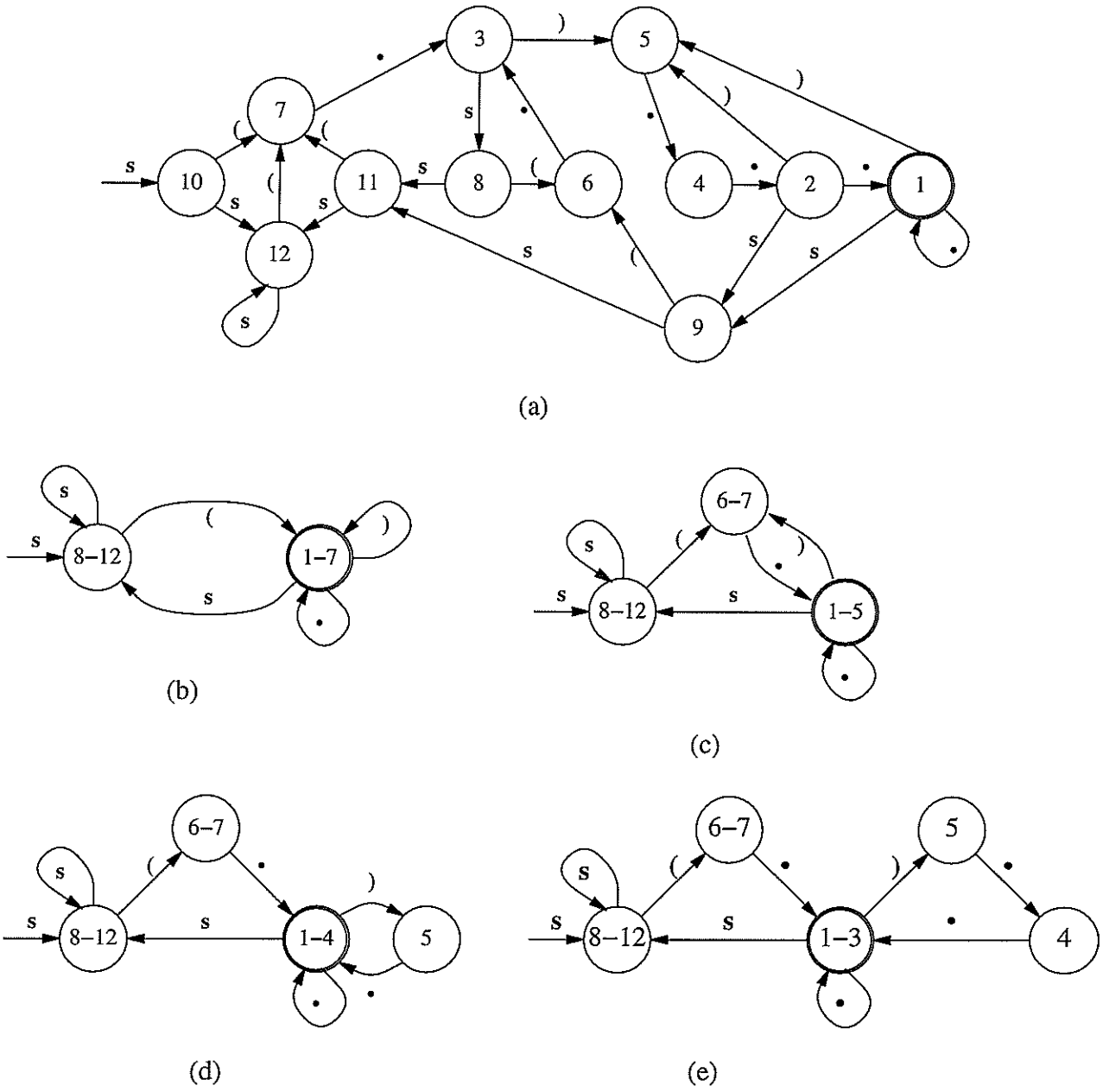


Figure 7. Twelve state finite-state machine (a) together with two (b), three (c), four (d) and five (e) state finite-state machines derived from clustering the parenthesis example patterns.

that. In this example, every state in every FSM with ≥ 4 states has the property that every transition leading to that state occurs under precisely the same member of the alphabet. In this way, clustering is showing that those patterns that need to produce a particular member of the alphabet upon decoding will be clustered together.

However, the decoding network must produce both a symbol and a distributed pattern.

Producing the correct symbol, as we have just discussed, is facilitated by the groupings that evolve from SRAAM training. Within each grouping, there is a more subtle structure carried by the patterns which aids the decoding network in producing the correct previous distributed pattern also. In the FSM version of the clustering, this corresponds to both being able to tell which symbol just occurred as well as what state just preceded the current state. In this way, the machine can be run in reverse and decoding can work iteratively to produce the encoded sequence.[4]

Further note that the more states we derive from clustering the more precise the finite-state acceptor becomes in relation to the training data. The language of the FSM of Figure 7 (b) is slightly different from that of Figure 7 (c), etc. While our one-state machine accepted everything, the two-state machine correctly fails to accept the string "s s s", and the three-state machine correctly fails to accept the string "s (", which the two-state machine accepts. The four-state machine reduces to the three-state machine, but the five state machine correctly fails to accept the string "s (•) •", which the three and four-state machines accept. As more states are added, the filtering of unacceptable strings gets more precise. We have already observed that the grammar describing the language of sequences is context-free and so the FSM is organized to approximate, as closely as possible, an acceptor for that language.

6.2. Clustering the syntactic parse tree representations

In our second example, it is much more difficult to view the clustered representations as an FSM since there are many more symbols in the language and fewer examples. Figure 8 shows the clustering results for the 25 tree representations, with each tree shown in its sequential form. In studying these results, two organizational strategies seem to predominate. First, clustering is based on the length of the sequence. Second, clustering is based on the most recently added symbols, i.e., the tail end of the sequence, analogous to our observations in the parenthesis language example. All sequences ending in four empty sub-trees (•) appear in the lower half of the cluster tree, while sequences ending in three or five empty sub-trees (•) appear in the upper half. While again an FSM could be constructed for this example, we choose to view it from the perspective of the size and shape of the trees themselves.

In Figure 9, it is perhaps easier to see how these properties are manifested. The trees are shown in clusters with a line separating major cluster groupings. The length of the sequence is reflected in the relative size of the tree, while the matching tail observation is related to shape.

[4] Note that in a theoretical sense, the reverse of any regular language (i.e., one in which each string is the reverse of a string from a regular language) is also regular. This is easily seen by reversing the direction of all transitions in the FSM to get a non-deterministic FSM and then transforming it into a deterministic one. SRAAM training must organize the representation space such that during encoding, symbols will cause transitions to occur from state to state, and during decoding the reverse transitions will occur. That is, there is no reorganization of the space to allow the reverse to work and so the space must be organized to work in both directions when trained. This requires that each n-dimensional point carry sufficient information to both produce the previous symbol and the previous state.

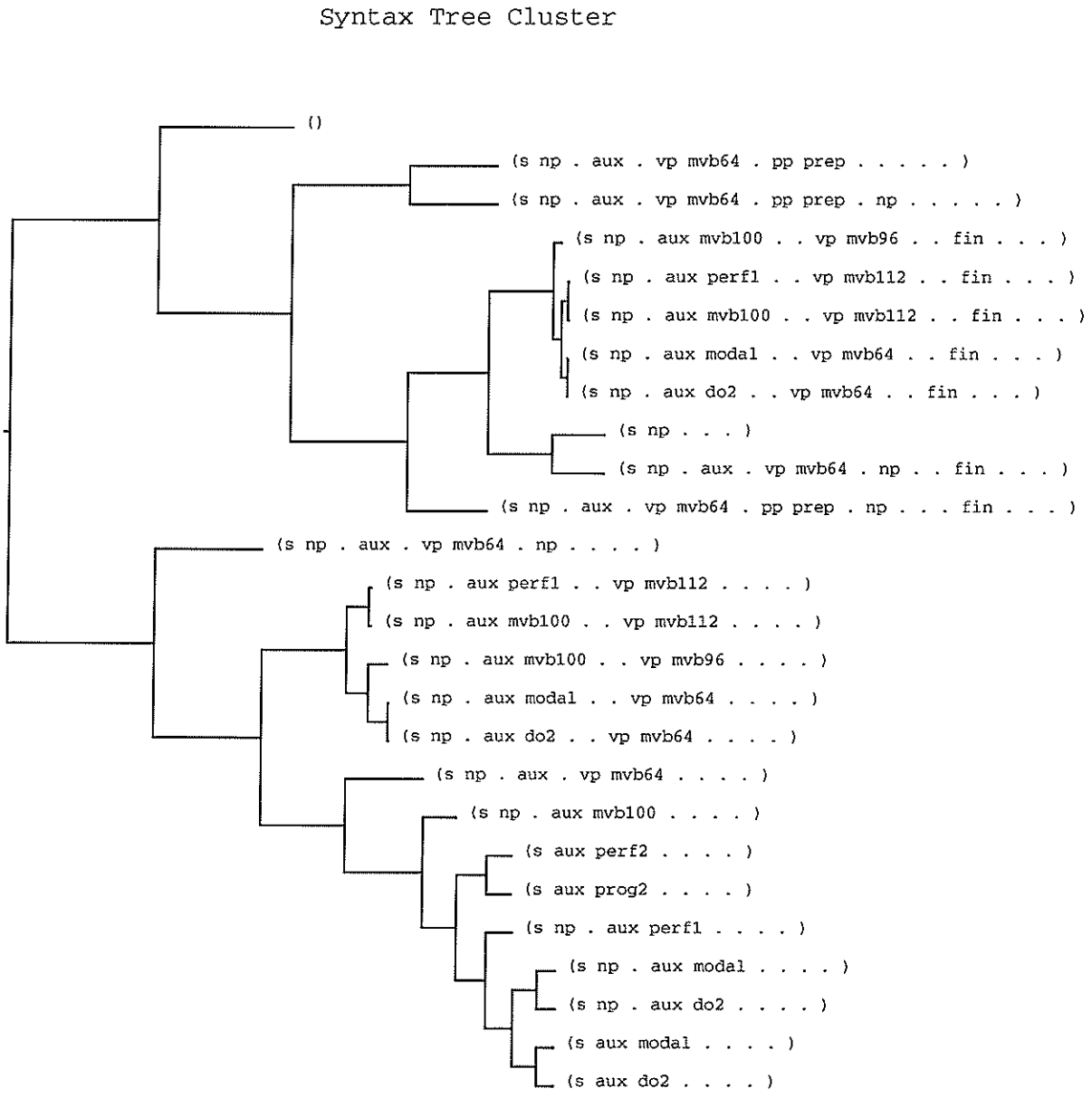


Figure 8. Cluster Analysis of Valid Tree Sequences.

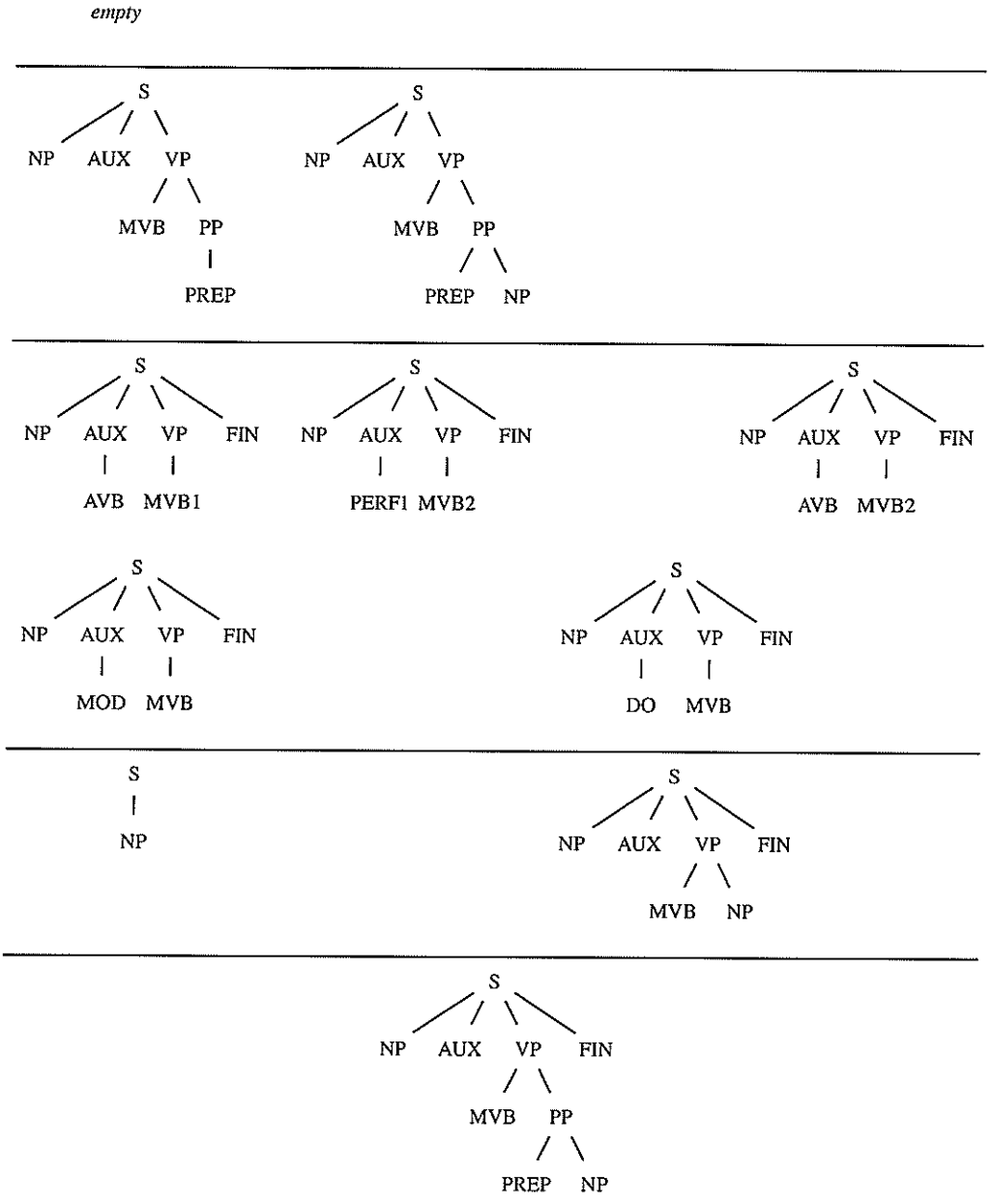


Figure 9a. Tree clusters — upper half of cluster tree.

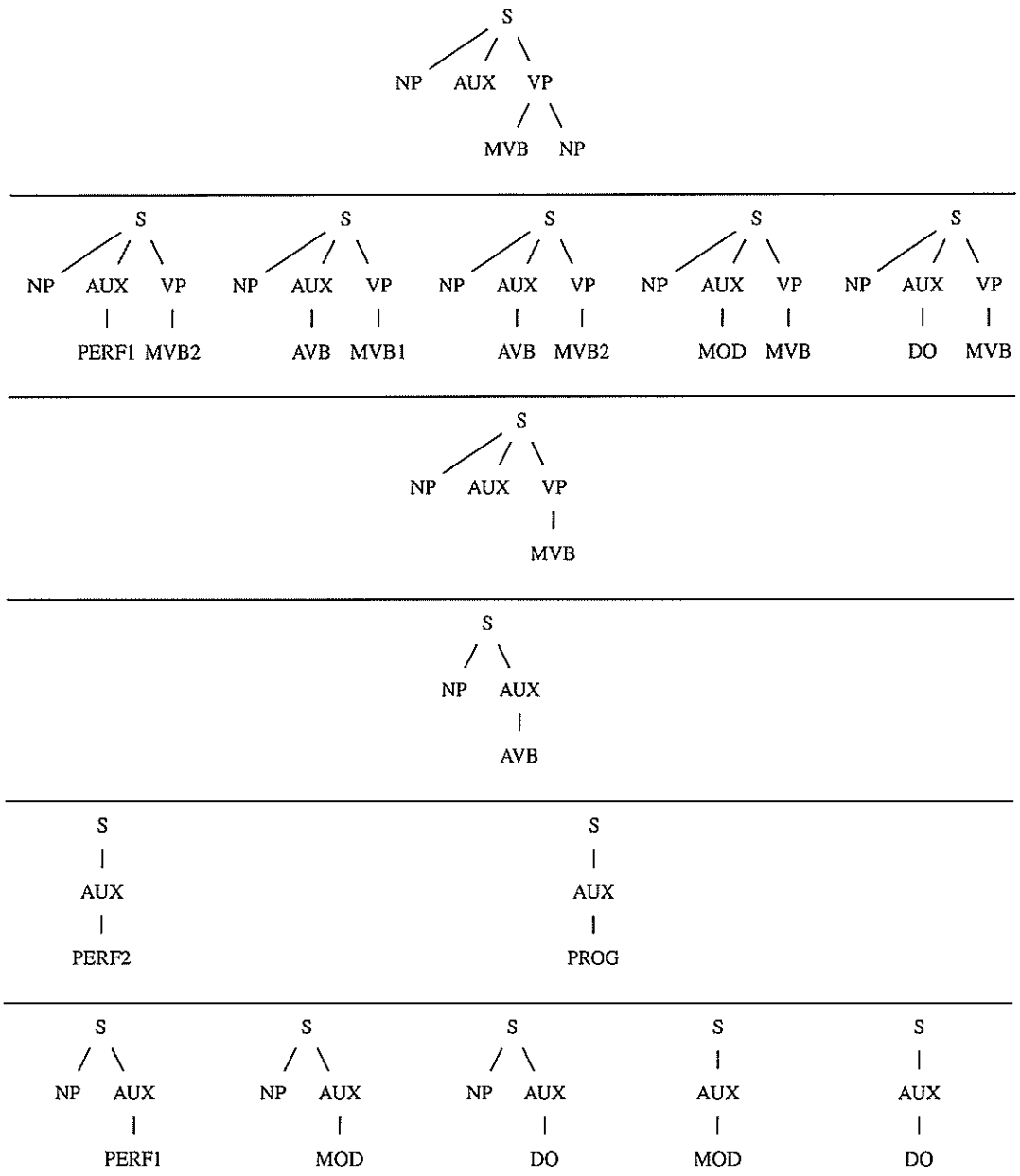


Figure 9b. Tree clusters — lower half of cluster tree.

7. Summary and Conclusions

When first introduced, RAAMs looked very promising as an answer to the compositionality issue for connectionism. However, they have several flaws. Training can be difficult and the valence of the structures to be represented must be known when deciding which network architecture to use. Internal node labeling in the structures is also prohibited.

Compositionality is easier under a sequential schema in which concatenation is a basic operation. Using SRAAMs and our slightly modified SRN training algorithm, the fixed

valence restriction disappears allowing each ordered forest of tree structures to be assigned a distributed representation. Training recurrent networks can be difficult, but training SRAAMs is much more straightforward, in our experience, than attempting to train RAAMs.

Given the removal of restrictions on shape and labeling of structures and given they are easier to train, RAAM networks take on new importance. With good generalization to demonstrate compositional properties, tail-recursive representations are worthy of further study.

8. Future Work

We are in the process of investigating better training strategies and other improvements to SRAAMs. In our parsing work, tail recursive distributed representations are being studied as possible targets for intermediate states in a deterministic parser based on recurrent networks. The ability to represent ordered forests, as opposed to fixed-valence hierarchies, opens up more possibilities. Simple tree structures should suffice for our purpose, but ordered forests may be useful for ambiguous sentences. Where ambiguity occurs, an ordered list of structural variations would provide a sequence of choices.

We have not fully explored the ability to encode multiple-tree forests in this paper. We expect further experimentation to demonstrate that representing populated forests works as well as the examples we used.

Further study is required to determine if the encoding of structure for SRAAM training will yield patterns that can be holistically transformed or probed for information contained within. Our results on this score were inconclusive. In the syntactic parse tree representations, we achieved 90% performance when attempting to train a non-recurrent network to report if a given symbol could be found within the parse tree that corresponded to a given distributed pattern. However, in the parenthesis example, we attempted to train a non-recurrent network to recognize if a given sub-tree, specified as a distributed pattern, was contained within another tree, also specified as a distributed pattern. Performance on novel examples for this experiment averaged in the low 60% range. A partial explanation for this can be seen from the family of FSMs. The final, accepting state contains a closely clustered set of patterns corresponding to all of the valid structures. Since these are grouped together, the task of the probe network is to extract very subtle properties from very similar patterns. Training is at odds with this since, as we have discussed earlier, clustering into states is a good strategy to satisfy the training goals.

Finally, Chrisman's work on dual-ported RAAMs may be perfectly compatible with SRAAM training. It also remains to be shown if a dual-ported variation of our sequential method can be trained to perform similar associations.

Acknowledgements

We thank the three anonymous reviewers who made helpful suggestions that improved this paper. We acknowledge and thank Andreas Stolcke for his version of the cluster program

used in this research. Thanks to Will Gillett who helped formulate parts of the discussion on FSMs and to Peter McCann who offered comments on drafts of this paper. We also thank Nancy Chang of Harvard University who worked on this project during the summer, 1992, as part of her Summer Undergraduate Research experience supported by NSF under Grant No. CDA-9123643.

References

Berg, George. (1992). A Connectionist Parser with Recursive Sentence Structure and Lexical Disambiguation. *Proceedings of AAAI-92, the Tenth National Conference on Artificial Intelligence*, 32–37.

Blank, D.S., Meeden, L.A., and Marshall, J.B. (1992). Exploring the Symbolic/Subsymbolic continuum: A Case Study of RAAM. In J. Dinsmore (Ed.) *Closing the Gap: Symbolism vs. Connectionism.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Chalmers, D.J. (1990). Syntactic Transformations on Distributed Representations. *Connection Science*, 2, 53–62.

Chrisman, Lonnie. (July, 1991). Learning Recursive Distributed Representations for Holistic Computation. Technical Report CMU-CS-91-154, Pittsburgh: School of Computer Science, Carnegie Mellon University.

Elman, Jeffrey L. (1991). Distributed Representations, Simple Recurrent Networks, and Grammatical Structure. *Machine Learning*, 7, 195–225.

Elman, Jeffrey L. (1990). Finding Structure in Time. *Cognitive Science*, 14, 179–212.

Kalman, B.L., and Kwasny, Stan C. (1993). TRAINREC: A System for Training Feedforward & Simple Recurrent Networks Efficiently and Correctly. Technical Report WUCS-93-26, St. Louis: Department of Computer Science, Washington University.

Kalman, B.L., and Kwasny, Stan C. (1991). A Superior Error Function for Training Neural Nets. *Proceedings of the International Joint Conference on Neural Networks*, Vol 2, Seattle, Washington, 40–52.

Kalman, B.L. (1990). Super Linear Learning in Back Propagation Neural Nets. Technical Report WUCS-90-21, St. Louis: Department of Computer Science, Washington University.

Knuth, Donald E. (1973). *The Art of Computer Programming.* Volume 1: Fundamental Algorithms. Reading MA: Addison-Wesley.

Kwasny, Stan C., Kalman, Barry L., and Chang, Nancy. (1993). Distributed Patterns as Hierarchical Structures. *Proceedings of the World Congress on Neural Networks*, Portland,

OR, Volume II, 198–201.

Kwasny, Stan C., and Kalman, Barry L. (1992). *A Recurrent Deterministic Parser.* Proceedings of the Fourth Midwest Artificial Intelligence and Cognitive Science Society Conference. Huntsville, AL: Intergraph Corporation.

Kwasny, Stan C., and Faisal, Kanaan A. (1992). Symbolic Parsing Via Subsymbolic Rules. In J. Dinsmore (Ed.) *Closing the Gap: Symbolism vs. Connectionism.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Plate, Tony A. (1991). Holographic Reduced Representations. Technical Report CRG-TR-91-1, Toronto: Department of Computer Science, University of Toronto.

Pollack, Jordan. (1989). Implications of Recursive Distributed Representations. In David S. Touretzky (Ed.) *Advances in Neural Information Processing Systems.* Los Gatos, CA: Morgan Kaufmann.

Pollack, Jordan. (1990). Recursive Distributed Representations. *Artificial Intelligence*, 46, 77–105.

Sperduti, Alessandro. (May, 1993). Labeling RAAM. TR-93-029, Berkeley, CA: International Computer Science Institute.

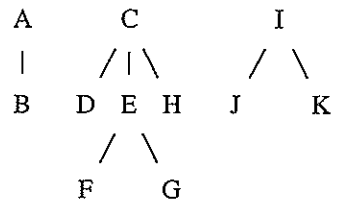
Sperduti, Alessandro. (June, 1993). On Some Stability Properties of the LRAAM Model. TR-93-031, Berkeley, CA: International Computer Science Institute.

Stolcke, Andreas, and Wu, Dekai. (April, 1992). Tree Matching with Recursive Distributed Representations. TR-92-025, Berkeley, CA: International Computer Science Institute.

APPENDIX A:
Algorithms for mapping between Forests, Binary Trees, and Sequences

In this appendix we provide routines in Scheme for mapping between a forest of trees, a binary tree, and a sequence. A more detailed description can be found in Knuth (1973). Four routines are required: two for mapping between forest and binary tree and two for mapping between binary tree and sequence. For the purpose of these algorithms, a forest is a list of general trees and a general tree is represented as a list whose first element is the label of the root and whose remaining elements are lists representing each subtree. A binary tree is either empty, represented as an empty list, or a three-element list whose first element is the label of the root and whose second and third elements are representations of the left and right subtrees respectively. A sequence of symbols consists of n symbols, for arbitrary n, and n + 1 dots (shown as |.| in Scheme).

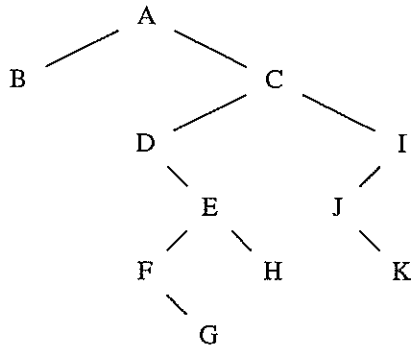
As an example, consider the following forest of trees:



In Scheme, we can represent the forest as nested list structures to give:

((A(B)) (C(D)(E(F)(G))(H)) (I(J)(K)))

Conversion to a single binary tree yields:



In Scheme, we can represent the binary tree as nested list structures as:

(A(B()())(C(D()(E(F()(G()()))(H()()))))(I(J()(K()()))()))

Conversion of this binary tree to a sequence yields:

A B |.| |.| C D |.| E F |.| G |.| |.| H |.| |.| I J |.| K |.| |.| |.|

The above translation from forest to binary tree to sequence is completely reversible.

Algorithm in Scheme for translating a forest of trees into an equivalent binary tree:

```
(define forest->binary-tree
;; converts a forest of general tree structures to a binary tree.
;; argument is a forest of general trees
;; returns a binary tree
 (lambda (trees)
  (if (null? trees)
     '()
     (let* ([tree (car trees)]
            [root (car tree)])
      (list root
          (forest->binary-tree (cdr tree))
          (forest->binary-tree (cdr trees)))))))
```

Algorithm in Scheme for translating a binary tree into an equivalent forest of trees:

```
(define binary-tree->forest
;; converts a binary tree into a forest of general trees.
;; argument is a binary tree
;; returns a forest of general trees
 (lambda (bintree)
  (if (null? bintree)
     '()
     (append
      (list
       (cons (car bintree)
             (binary-tree->forest (cadr bintree))))
      (binary-tree->forest (caddr bintree))))))
```

Algorithm in Scheme for translating a binary tree into an equivalent sequence of symbols:

```
(define binary-tree->sequence
;; converts a binary tree to a list representation of a binary tree.
;; argument is a binary tree
;; returns a sequence of symbols and dots.
 (lambda (bintree)
  (if (null? bintree)
     (list '|.|)
     (let ([root (car bintree)]
           [left (cadr bintree)]
           [right (caddr bintree)])
      (cons root
          (append
           (binary-tree->sequence left)
           (binary-tree->sequence right)))))))
```

Algorithm in Scheme for translating a sequence of symbols into an equivalent binary tree:

```
(define sequence->binary-tree
;; convert list of atoms representing a binary tree into a binary tree.
;; a binary tree consists of a list of three components:
;; the root
;; the left binary subtree, or ()
;; the right binary subtree, or ()
;; argument is a list of atoms such that for n symbols there are (n+1) dots.
 (lambda (lat)
  (letrec ([lstpos lat]
        [iterate
         (lambda ()
          (if (null? lstpos)
             '()
             (let ([nxt-atm (car lstpos)]
                   [rest (cdr lstpos)])
              (cond ((equal? nxt-atm '|.|)
                          (set! lstpos rest)
                          '())
                    (else
                          (set! lstpos rest)
                          (list nxt-atm (iterate) (iterate)))))))])
   (iterate))))
```