INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

Labeling RAAM

Alessandro Sperduti

TR-93-029

May 1993

Abstract

In this paper we propose an extension of the Recursive Auto-Associative Memory (RAAM) by Pollack. This extension, the Labeling RAAM (LRAAM), is able to encode labeled graphs with cycles by representing pointers explicitly. A theoretical analysis of the constraints imposed on the weights by the learning task under the hypothesis of perfect learning and linear output units is presented. Cycles and confluent pointers result to be particularly effective in imposing constraints on the weights. Some technical problems encountered in the RAAM, such as the termination problem in the learning and decoding processes, are solved more naturally in the LRAAM framework. The representations developed for the pointers seem to be robust to recurrent decoding along a cycle. Data encoded in a LRAAM can be accessed by pointer as well as by content. The direct access by content can be achieved by transforming the encoder network of the LRAAM in a Bidirectional Associative Memory (BAM). Different access procedures can be defined according to the access key. The access procedures are not wholly reliable, however they seem to have a high likelihood of success. A geometric interpretation of the decoding process is given and the representations developed in the pointer space of a two hidden units LRAAM are presented and discussed. In particular, the pointer space results to be partitioned in a fractal-like fashion. Some effects on the representations induced by the Hopfield-like dynamics of the pointer decoding process are discussed and an encoding scheme able to retain the richness of representation devised by the decoding function is outlined. The application of the LRAAM model to the control of the dynamics of recurrent high-order networks is briefly sketched as well.

1 Introduction

In the last years, different researchers have focused their efforts to demonstrate how symbolic structures such as lists, trees, and stacks can be represented and manipulated in a connectionist system while preserving all the computational characteristics of the connectionism (and extending them to the symbolic representations). The goal of these researchers is to provide evidence of the potentiality of the connectionist approach to handle domains of structured tasks. The common background of their ideas is the search for a realization of the distal access ability and consequently of the compositionality one. BoltzCONS [Tou90] is an example of how a connectionist system (i.e. Boltzman machine) can handle symbolic structures. It is based on parallel associative retrieval and it differs from other connectionist systems because it constructs and modifies composite symbol structures dynamically, by representing them as activity patterns rather than as weights. It uses distributed representations of linked lists, loaded in coarse-coded memories [1] [RT87], as basic representational elements and LISP's *car*, *cdr*, and *cons* functions as basic operations. Links are implemented by associations. The associative retrieval capabilities of BolzCONS support computational primitives such as instantaneous access to an element of a list or the capability to rapidly access parts of a symbol structure based on closest match (rather than symbolic exact match) or efficient retrieval using multiple cues (in general, supplying more constraints causes a connectionist model to settle faster; whereas a conventional machine using hash table representations will be slowed down). The full architecture supports direct representations of arbitrary tree structures, and it can perform complex pointer manipulations using multiple buffers operating simultaneously on its memory. It must be noted that the dynamic manipulation of structures by BoltzCONS requires the same allocation and reclamation problems usually met in conventional symbolic systems. Moreover, even if coarse-coding representations give successful results, there remain some problems associated with this representation strategy. Firstly, coarse-coding requires expensive and complex access mechanisms (i.e. pullout networks). Secondly, coarse-coded memories can only simultaneously instantiate a small number of representational elements before spurious ones are introduced. Finally, the coarse-coded memory of BoltzCONS needs a huge number of units because it utilizes binary codes.

The above problems, together with the observation that BoltzCONS needs a large amount of human effort to design, to compress and to tune the representations, have stimulated Pollack [Pol90] to design the Recursive Auto-Associative Memory architecture (RAAM). The RAAM system uses back-propagation to discover compact recursive distributed representations for fixed-valence trees. The compact recursive distributed representations obtained are very interesting because they synthesize the characteristics of an item (categorical features) preserving its individual peculiarity (distinctive features). Moreover they utilize real values over few units and they can be composed selectively by a potentially very large number of primitive elements. Other advantages are related to the fact that they are developed mechanically, their aggregation mode is compositional, and their access mechanisms are simple and deterministic. However the most relevant aspect of this work is that a RAAM can devise representations of trees as numeric vectors. If

[1]In a coarse-coded memory each unit participates in the representation of many entities. A unit of such a memory is said to be coarsely tuned.

inference over trees might be performed by numerical transformation (i.e. neural networks) over their numerical representation, very fast and cheap inference engines would be built (see [Cha90]).

A more formal characterization of representations of structures in connectionist systems has been developed by Smolensky [Smo90]. He reduces the problem of representing structured objects to three subproblems: decomposing the structures via roles, representing conjunctions, and representing variable/value bindings. The representation of variable/value bindings is obtained through tensor algebra. In the tensor product representation, both the variables and the values can be arbitrarily distributed [2] , enabling representations in which every unit is part of the representation of every constituent in the structure. This generality allows to express existing cases of connectionist representations as particular cases of the tensor representations. Smolensky discusses several features of the tensor product representation, such as the graceful saturation and exact retrieval for non saturated memories, the representation of continuous structures as well as of finite ones, the representation of symbolic operators and recursive structures, the possibility to define and analyze optimal role vectors, and to use a value bounded to one variable as a variable.

Reduced representations of structured objects in connectionist systems are related by Hinton to the problem of mapping part-whole hierarchies into connectionist networks [Hin90]. The scheme he proposes considers two quite different methods for performing inference. Simple "intuitive" inferences can be performed by a single settling of a network without changing the way in which the world is mapped into the network. More complex "rational" inferences involve a sequence of such settlings with mapping changes after each settling. He discusses three approaches to map a part-whole hierarchy into a finite amount of parallel hardware: fixed mappings, within-level timesharing, and between-level timesharing. In the first approach each object in the hierarchy is always mapped into a pattern of activity in the same set of units, and each set of units is always used to represent the same object (one-to-one mapping). In the second approach, many different objects at the same level can be mapped into the same set of units in the serial recognition apparatus, but whenever one of these objects is recognized, it is represented in the same units. Finally, between-level timesharing allows many different objects at the same level to be mapped into different sets of units depending on the level at which attention is focused. The idea that he proposes is to use role-specific reduced representations to implement pointers. This kind of pointers, as Pollack's reduced distributed representations, contains relevant information about the object they point to and so some type of computations over objects can be performed on them, without accessing to the full description of the object. In this framework a general computation results from the combination of slow sequential access to full descriptions and fast parallel constraint-satisfaction using reduced descriptions ("intuitive" inference). If the appropriate representations are known, an "intuitive" inference allows to make a lot of useful computation. When the computational task is difficult, a sequence of settlings is executed, and after each settling the mapping between the world and the network is changed. The possibility to change the mapping allows to apply the knowledge of the system to any part of the task.

[2]The choice to have distributed representations of variables (i.e. roles) is controversial. Hinton [Hin90] believes that in a nonlinear system it is probably easier to make use of the information about the fillers of roles if this information is localized.

In this paper, we present an extension of the RAAM, the Labeling RAAM (LRAAM). A LRAAM allows to store a label for each component of the structure to be represented, so to generate reduced representations of labeled graphs. Moreover, data encoded in a LRAAM can be accessed not only by pointer but also by content.

In Section 2 we review the standard RAAM model and discuss some technical problems encountered in this model. The proposed extension is discussed in Section 3, where a theoretical analysis of the constraints on the weights matrix imposed by the learning task under some particular conditions is presented. On the basis of this analysis some suggestions on how to represent the training set are given. A more natural approach to some technical problems of the RAAM model, such as the termination problem of the decoding process and the use of double tolerances, is discussed as well and some examples of encoding of single structures are shown.

The possibility to access data not only by pointer, but also by content is discussed in Section 4. In this section we show how transforming the encoder network used by the LRAAM in a Bidirectional Associative Memory (BAM), it is possible to access directly data by content with a high likelihood of success.

A study of the representational space built by a two hidden unit LRAAM is presented in Section 5. More structures are present in the training set. The decoding function of the LRAAM is analyzed and the set of structures found to be representable by the pointer space discussed on the basis of a geometric interpretation of the former.

A discussion of the main characteristics of the LRAAM model, compared with the RAAM model and the Hopfield model, is presented in Section 6. In particular, an example of how LRAAM can be used to control the dynamics of a recurrent network is briefly presented. Conclusions are stated in Section 7.

2 The RAAM model

The RAAM (Recursive Auto-Associative Memory) was introduced by Pollack [Pol90, Pol89] to allow traditional symbolic data structures, such as trees with labeled leaves, to be represented subsymbolically as distributed patterns of activation. In the last years, different papers have discussed or used the RAAM model with interesting results [Chr91, BMM92, Rei92, SW92].

The basic RAAM can encode arbitrary tree structures of variable depth but fixed branching factor (valence). The idea is to map a symbolic tree into a numeric vector and then to reconstruct a very close approximation of the symbolic tree starting from the numeric vector. The mapping from trees to numeric vectors can be obtained by a *compressor* able to encode small sets of fixed-width numeric patterns into single patterns of the same size. The compressor can be applied starting from the leaves of the tree back to the root in a recursive fashion, obtaining at the end of the process a fixed-width pattern representing the entire structure. For example, consider the simple binary tree $((A\ B)(C\ D))$, shown in Figure 1, where A, B, C, D are of equal size.

Firstly, A and B are compressed into a pattern, R_1, and C and D into another pattern, R_2. Then R_1 and R_2, are compressed into the pattern R_3 which represents the whole tree. The tree represented by R_3 can then be reconstructed in a top down fashion by a *reconstructor* which is able to decode a compressed representation back to its components.

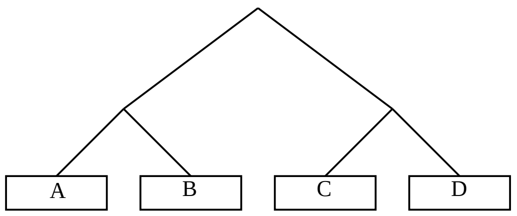

Figure 1: Example of binary tree.

Thus, at the first step, R_3 is decoded into R'_1 and R'_2. Then R'_1 is decoded into A' and B', and R'_2 into C' and D'. The *compressor* and *reconstructor* may be realized together by an *Encoder Network* (see Figure 2). Usually it is trained using back-propagation on a static training set until the network is able to reproduce on the output layer the same pattern as on the input layer by first encoding the input into an internal representation across the hidden layer, and then decoding this hidden representation back to the original pattern on the output layer. In the RAAM a dynamic training set is allowed to face the recursive nature of the task at hand. In a general RAAM (see Figure 2) the number of units in the input (output) layer is a multiple of the number of hidden units.

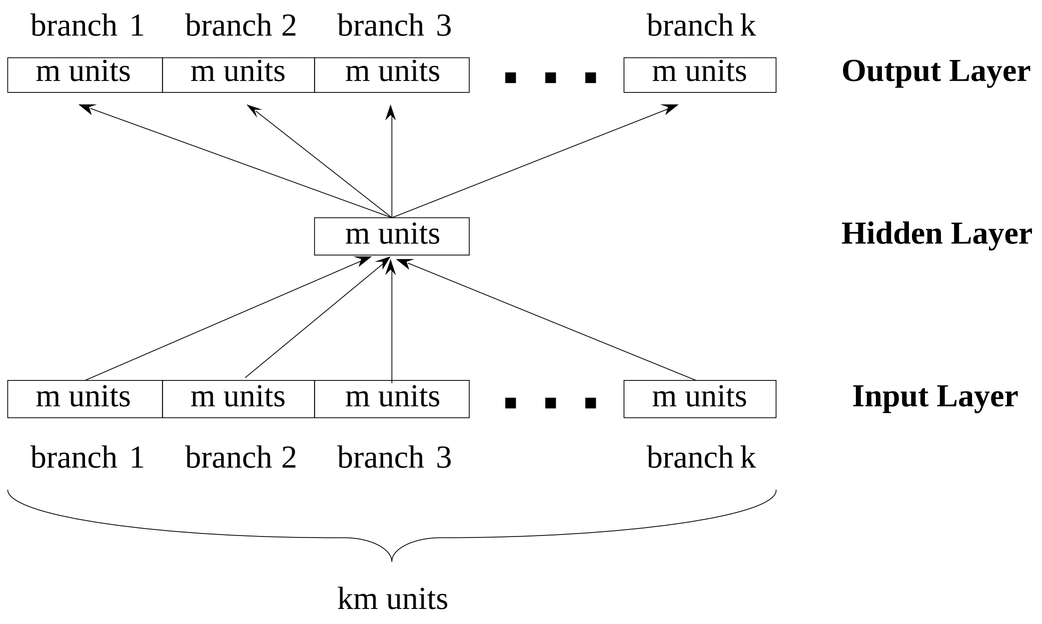

Figure 2: The encoder network for a general *RAAM*

More precisely, if the valence of the tree which must be represented is k, and each symbol in the tree is represented with m units, then the network must have m hidden units and km input (output) units. This constraint on the network is introduced since a group of m units in the input (output) layer must be used, at different times, both to represent the labels on the leaves and the compressed representations obtained by them on the hidden layer. In the case of the example given above we have $k = 2$. Thus the *Encoder Network* is a $2m - m - 2m$ feed-forward network and it must be trained as follows:

input pattern		hidden pattern		output pattern
$(A\ B)$	$\rightarrow$	$R_1(t)$	$\rightarrow$	$(A'(t)\ B'(t))$
$(C\ D)$	$\rightarrow$	$R_2(t)$	$\rightarrow$	$(C'(t)\ D'(t))$
$(R_1(t)\ R_2(t))$	$\rightarrow$	$R_3(t)$	$\rightarrow$	$(R'_1(t)\ R'_2(t))$

where t represents the time, or epoch, of training. If the back-propagation algorithm is able to learn the training set perfectly (perfec learning), i.e., the total error goes to zero, it can be stated that:

$$\begin{aligned} A' &= A \\ B' &= B \\ C' &= C \\ D' &= D \\ R'_1 &= R_1 \\ R'_2 &= R_2. \end{aligned}$$

Note that at the end of learning we obtain not only a distributed representation of the whole tree, but also of each subtree. In fact, R_1 is a compressed representation of $(A\ B)$ and R_2 of $(C\ D)$.
A version of the RAAM, called *Sequential RAAM (SRAAM)*, can be used also to encode sequences. In fact, sequences such as $(A\ B\ C)$ can be represented as left-branching binary trees, i.e., $(((NIL\ A)\ B)\ C)$. An SRAAM allows a *Last-In-First-Out* access mechanism for sequences. The architecture of an SRAAM (see Figure 3) is simpler than a RAAM, since only a branch of the tree is used.

The input (output) layer is split in two groups of units: one group is used to represent the *top* of the sequence and the other the compressed representation of the *stack*; obviously, the hidden layer counts as many units as the representation of the stack in the input layer. One advantage of the SRAAM is that the number of units used in the representation of the stack can be larger than the one used in representing the top, which allows to size up the dimension of the hidden layer according to the length of the sequence. The training set for the sequence above is as follows:

input pattern		hidden pattern		output pattern
$(NIL\ A)$	$\rightarrow$	$R_a(t)$	$\rightarrow$	$(NIL'(t)\ A'(t))$
$(R_a(t)\ B)$	$\rightarrow$	$R_{ab}(t)$	$\rightarrow$	$(R'_a(t)\ B'(t))$
$(R_{ab}(t)\ C(t))$	$\rightarrow$	$R_{abc}(t)$	$\rightarrow$	$(R'_{ab}(t)\ C'(t))$

After convergence, R_{abc} will be a representation for $(A\ B\ C)$, R_{ab} for $(A\ B)$, and R_a for (A). As a final observation, note that, since the encoding strategy of the RAAM is based on associative mechanisms, a single RAAM (resp., SRAAM) may encode a single tree (resp., sequence) as well as a collection of trees (resp., sequences).

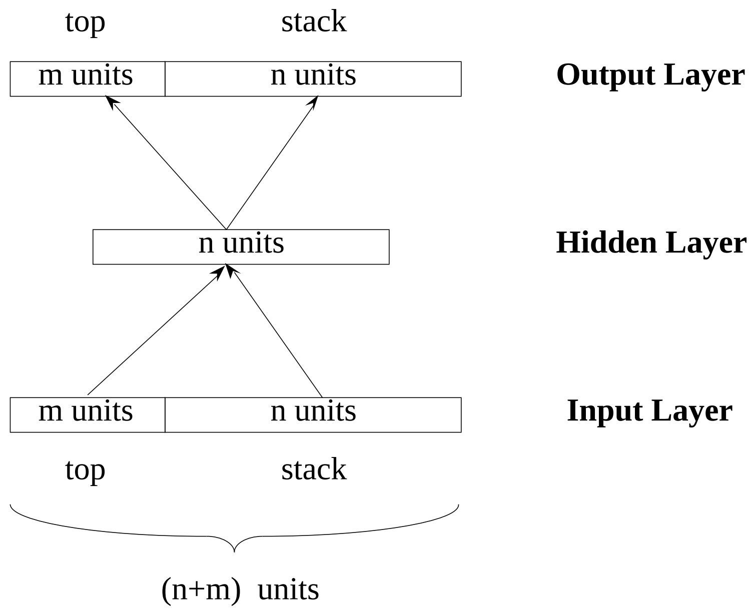

Figure 3: The encoder network for an $SRAAM$.

2.1 Technical Problems

The procedure described in the previous section to synthesize a RAAM presents some technical problems.

The first problem concerns learning. The fact that the training set is dynamic leads to the so called *Moving Target Problem*, i.e., the patterns which are not terminals change during learning and consequently also the targets of the network change. Since, in general, the nonterminals constitute about half of the training set, the resulting mobility of the target may invalidate the convergence of learning. To prevent this event, small learning parameters must be used, so that changes in the hidden representations do not overcome the decreasing error granted by the changes in weights. Learning parameters, however, must be large enough to allow learning. As a consequence of such setting, learning is, in general, rather slow.

Another problem with learning concerns the stopping criteria. Since it is not realistic to expect perfect learning in a finite time using back-propagation, some tolerance measure must be used. In general, the terminals are represented by binary patterns, whereas nonterminals are real-valued patterns since they represent activation patterns of the hidden layer. Because of this difference between terminals and nonterminals, it is convenient to have two different tolerances according to the type of pattern at hand. Terminal patterns may have a loose tolerance, τ, (conventionally set to 0.2), whereas nonterminal patterns need a more strict tolerance, ν (conventionally set to 0.05). Setting these tolerances to the correct values is very important, since the ability of the network to reconstruct the encoded tree depends heavily on them, especially when the encoded trees are deep.

A termination problem is also present in the decoding of a compressed representation. How to decide if a decoded pattern is a terminal or a nonterminal? A way to solve the problem is to perform a test for "binary-ness" which consists in checking if all the values of a pattern are above $1 - \tau$ or below τ. However, it may happen that a nonterminal passes as well the test for "binary-ness". Even if we can prevent this event by bounding the range of activation of the hidden units, thus introducing a further constraint to the already difficult task imposed by the moving target, the solution is not feasible if the terminals are real-valued. A more robust solution would be to train a classifier to discriminate between terminals and nonterminals, however it would result in a relevant computational overload. A more elegant solution to this problem was developed by Stolcke and Wu [SW92]. They used one unit of the hidden layer to represent explicitly the distinction between terminals and nonterminals. In order to obtain this distinction, they injected an extra error in one of the units of the hidden layer during learning, forcing the output of the unit to be 1 for all nonterminal codes, and 0 for all terminal ones. Actually, this method, which we call the *Injection Method*, corresponds to develop a built-in classifier during learning. The advantage of having a built-in classifier with respect to an external one is that the internal representations of the RAAM may change in such a way to simplify the computational task of the classifier, i.e., the classification problem becames linearly separable. Obviously, the injection method implies harder constraints on the learning task of the encoder as well.

In the next section we will show how the introduction of a new variant of the RAAM, the *Labeling RAAM (LRAAM)*, which enables to store a label for each node of the tree (terminal or nonterminal), allows one not only to have a more powerful tool to encode complex structures, but also to give more satisfactory solutions to these technical problems.

3 Labeling RAAM

In this section we introduce a simple variant of the RAAM, the *Labeling RAAM (LRAAM)*. It differs from the RAAM because it allows us to encode labeled structures. The general structure of the encoder network for an LRAAM is shown in Figure 4.

The idea is to allocate a part of the input for the label and the rest for one or more pointers. The pointer fields must be all equal in size and of the same dimension of the hidden layer. Actually, an LRAAM is a generalization of an SRAAM. In fact, an SRAAM is an LRAAM with only one pointer field.

An LRAAM has several advantages over a standard RAAM. Firstly, it is more powerful, since it allows to encode directed graphs where each node has a bounded number of outgoing arcs. Secondly, the dimension of the label needs not to be equal to the dimension of the pointers, thus allowing one to size up the dimension of the hidden layer according to the complexity of the structure at hand. Moreover, a part of the label can be used to indicate if the pointers are void or not, thus solving the termination problem in the decoding phase. Lastly, we show in the next section how an LRAAM allows direct access to the components of the encoded structure not only by pointer, but also by content.

3.1 Encoding of graphs

Labeled graphs can be easily encoded using an LRAAM. It suffices to represent each node of the graph as a record with one field for the label and one field for each pointer to a connected node. The pointers need to be only logical pointers, since their actual values will

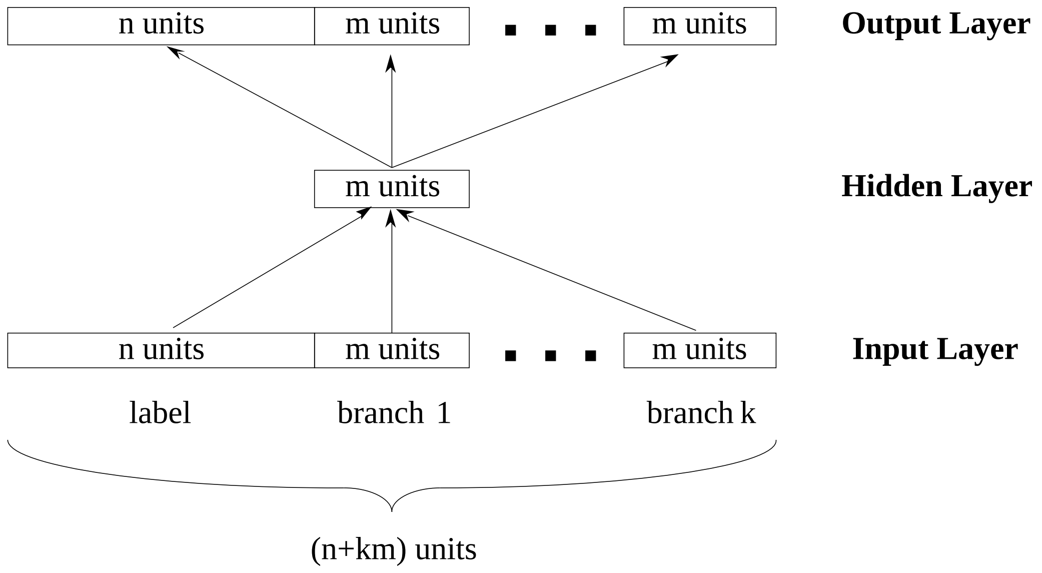

Figure 4: The encoder network for a general $LRAAM$.

be computed by learning. A graph will be represented by a list of such records, and such a list will be the training set for the LRAAM. For example, one way to represent the graph shown in Figure 5 would be as follows:

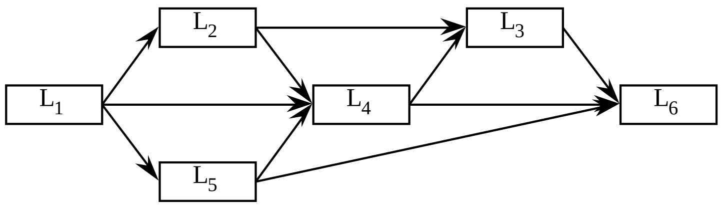

Figure 5: Example of graph.

input		hidden		output
$(L_1\ P_{n2}\ P_{n4}\ P_{n5})$	$\rightarrow$	$P_{n1}(t)$	$\rightarrow$	$(L'_1(t)\ P'_{n2}(t)\ P'_{n4}(t)\ P'_{n5}(t))$
$(L_2\ P_{n3}\ P_{n4}\ NIL)$	$\rightarrow$	$P_{n2}(t)$	$\rightarrow$	$(L'_2(t)\ P'_{n3}(t)\ P'_{n4}(t)\ NIL'(t))$
$(L_3\ P_{n6}\ NIL\ NIL)$	$\rightarrow$	$P_{n3}(t)$	$\rightarrow$	$(L'_3(t)\ P'_{n6}(t)\ NIL'(t)\ NIL'(t))$
$(L_4\ P_{n6}\ P_{n3}\ NIL)$	$\rightarrow$	$P_{n4}(t)$	$\rightarrow$	$(L'_4(t)\ P'_{n6}(t)\ P'_{n3}(t)\ NIL'(t))$
$(L_5\ P_{n4}\ P_{n6}\ NIL)$	$\rightarrow$	$P_{n5}(t)$	$\rightarrow$	$(L'_5(t)\ P'_{n4}(t)\ P'_{n6}(t)\ NIL'(t))$
$(L_6\ NIL\ NIL\ NIL)$	$\rightarrow$	$P_{n6}(t)$	$\rightarrow$	$(L'_6(t)\ NIL'(t)\ NIL'(t)\ NIL'(t))$

where L_i and P_{ni} are respectively the label and the pointer to the i-th node. For the sake of simplicity, the void pointer is represented by a single symbol, NIL, but it must be pointed out that each instance of it must be considered different. This statement will be made clear in Section 3.2.

An important question about the representation of a graph is which aspects of the representation itself can make the encoding task harder or easier. In order to get some knowledge about this problem, we discuss in the following a theoretical analysis on the constraints imposed by the representation on the set of weights of the LRAAM, under the hypotheses of perfect learning (zero total error after learning) and linear output units. The features we study are cycles and confluent pointers.

3.1.1 Cycles and confluent pointers

Cycles are the first feature of a graph which one can expect to impose strong constraints on the set of weights of an LRAAM. In order to give an idea about these constraints, consider a single pointer field in the output layer and the occurrence of the simplest possible cycle, i.e., two nodes, N_1 and N_2, pointing to each other:

$$N_1 \leftrightarrow N_2. \tag{1}$$

If we name W_p the weight matrix of the connections outgoing from the hidden units and entering the output units representing the pointer field, then, after learning, the pointers to the nodes, $\vec{p}_1$ and $\vec{p}_2$, must satisfy the following constraints:

$$W_p\vec{p}_1 = \vec{p}_2, \tag{2}$$
$$W_p\vec{p}_2 = \vec{p}_1, \tag{3}$$

which implies that:

$$W_p^2\vec{p}_1 = \vec{p}_1, \tag{4}$$
$$W_p^2\vec{p}_2 = \vec{p}_2. \tag{5}$$

In general, given a cycle of length k, the following system of equations must be satisfied:

$$W_p^k\vec{p}_j = \vec{p}_j,\ j = 1, \cdots, k. \tag{6}$$

This means that pointers to nodes belonging to the same cycle and represented in the same pointer field, must be eigenvectors of the matrix W_p^k. Moreover, the eigenvalues corresponding to those eigenvectors are all equals to 1. It can also be verified directly that if $\vec{x} = \sum_{j=1}^{k}\vec{p}_j \neq \vec{0}$, where $\vec{0}$ is the null vector, then $\vec{x}$ is an eigenvector of the matrix W_p corresponding to a unity eigenvalue.

Other constraints can be imposed on W_p, and also on the pointer representations, by *confluent pointers (with respect to a pointer field)*, i.e., pointers to the same node represented in the same pointer field. Suppose, for example, that nodes N_1 and N_2 point to the same node N_3 then we have:

$$W_p\vec{p}_1 = \vec{p}_3, \tag{7}$$
$$W_p\vec{p}_2 = \vec{p}_3, \tag{8}$$

which implies that W_p cannot be of full rank, since $\vec{p}_1 \neq \vec{p}_2$. Note that $\vec{p}_1$ must be linearly independent with respect to $\vec{p}_2$ to satisfy the above equations. However if another node N_4 points to N_3, then the pointer to N_4, $\vec{p}_4$, may be either linearly independent with respect to $\vec{p}_1$ and $\vec{p}_2$, or a linear combination of them of the kind:

$$\vec{p}_4 = \alpha\vec{p}_1 + (1-\alpha)\vec{p}_2, \tag{9}$$

where α is any real number. The general form of the equation above is:

$$\vec{p}_t = \sum_{j=1}^{t-2}(\alpha_j\vec{p}_j) + (1 - \sum_{j=1}^{t-2}\alpha_j)\vec{p}_{t-1}, \tag{10}$$

where α_j are real numbers and $\vec{p}_j$ $(j = 1, \cdots, t-1)$ are linearly independent. Actually, the equation states that $\vec{p}_t$ must be a point on the hyperplane defined by the vectors $\vec{p}_j$ $(j = 1, \cdots, t-1)$. The constraints imposed on one pointer by the label and the other pointer fields will force one of the two solutions. For example, supposing no other pointer fields, if W_l is the weights matrix of the connections outgoing from the hidden units and entering in the output units representing the label, the following equations must be satisfied:

$$W_l\vec{p}_1 = \vec{l}_1, \tag{11}$$
$$W_l\vec{p}_2 = \vec{l}_2, \tag{12}$$

where $\vec{l}_1$ and $\vec{l}_2$ are the labels of N_1 and N_2. Because of these equations, if $\vec{p}_4$ is a linear combination of $\vec{p}_1$ and $\vec{p}_2$, then it must be satisfied also the following equation:

$$\vec{l}_4 = W_l(\alpha\vec{p}_1 + (1-\alpha)\vec{p}_2) = \alpha\vec{l}_1 + (1-\alpha)\vec{l}_2, \tag{13}$$

where $\vec{l}_4$ is the label of N_4. Thus $\vec{l}_4$ must be linearly dependent on $\vec{l}_1$ and $\vec{l}_2$.

It must be pointed out that the analysis we made in this section is too restrictive with respect to the case of approximate learning and nonlinear output units. It, however, gives some hints on how to represent a graph in order to avoid unnecessary constraints on the learning task.

3.1.2 Graph representation

In the previous section we have shown that cycles represented in the same pointer field and confluent pointers impose strong constraints on the structure of the weight matrix. In some cases, when several pointer fields are present and no particular role is attached to them, the number of confluent pointers to a node depends on the representation of the graph. For example, the representation that we gave at the beginning of the graph shown in Figure 5 presents two cases of confluent pointers: nodes N_1 and N_2 point to node N_4 (confluent with respect to the second pointer field) and nodes N_3 and N_4 to node N_6 (confluent with respect to the first pointer field). Since the NIL pointer can be considered as a "don't care" symbol (we will discuss this statement in the next subsection), and since we had no particular role, i.e., meaning, attached to the pointer fields, the confluence of the pointers can be removed by rearranging the pointers as follows:

$(L_1\ P_{n2}\ P_{n4}\ P_{n5})$
$(L_2\ P_{n3}\ NIL\ P_{n4})$
$(L_3\ NIL\ NIL\ P_{n6})$
$(L_4\ P_{n6}\ P_{n3}\ NIL)$
$(L_5\ P_{n4}\ P_{n6}\ NIL)$
$(L_6\ NIL\ NIL\ NIL)$

The rearrangement of the pointers, in this case, has also the advantage to balance the computational load over the matrices associated with the pointer fields. The same approach can be followed when dealing with cycles.

3.2 The void pointer problem

One advantage of LRAAMs over RAAMs is the possibility to solve the termination problem of the decoding phase by allocating one bit of the label for each pointer to represent if the pointer is void or not. This solution works better than to fix a particular pattern for the void pointer, such as a pattern with all the bits to 1 or 0 or -1 (if symmetrical sigmoids are used). Simulations performed with symmetrical sigmoids showed that the configurations with all bits equal to 1 or -1 were used also by non void pointers, whereas the configuration with all bits set to zero reduced considerably the rate of convergence. The use of a part of the label to solve the problem is particularly efficient since the pointer fields are free to assume every configuration when they are void, and this adds more degrees of freedom to the system. In order to avoid instabilities for the void pointers, their output activation at one epoch is used as input activation at the next epoch. Experimentation showed fast convergence to different fixed points for different void pointers. For this reason, in Section 3.1 we claimed that each occurrence of the void pointer is different and that the NIL symbol can be considered like a “don’t care” symbol.

Our opinion about this approach to the void problem, in the context of the LRAAM model, is that it compares favorably to the Injection Method, too. In fact, because of the multiple representations the void pointer gets, the occurrence of the same label at the leaves of different trees have a chance to be represented in different ways, even if the relevant information, i.e., the label value, is the same.

3.3 Single tolerance

We have seen that the training of a RAAM needs two tolerance parameters, one for the terminal patterns (which are binary), τ, and one for the nonterminal patterns (which are real-valued), ν. An advantage of an LRAAM, when binary labels are considered, is that units representing labels are different from units representing pointers, whereas in a RAAM a unit is used to represent both terminals and nonterminals. This fact allows us to remove the need for a double tolerance if the steepnesses of the sigmoids of the units can be modified at some point during learning. The basic idea is that units representing labels are more effective if they are saturated (high steepness), while units representing pointers are more effective if they are not saturated (low steepness), since they need to assume several different values. This consideration leads to the following additions to the learning procedure:

1. The steepnesses of output units representing labels are initialized to a value vo_l higher than the value vo_p used for output units representing pointers, and the steepness of the hidden units is initialized to a value $vh < vo_p$.

2. When the LRAAM is able to *map the labels correctly*, the steepnesses of the corresponding units are set to a value $VO_l \gg vo_l$.

3. Only the tolerance ν is used for every output unit.

The labels are said to be *correctly mapped* if for each training pattern and for each output unit representing the label holds that:

$$netl_i^{(p)} = \begin{cases} y > 0 & \text{if } l_i^{(p)} = \xi^+ \\ y < 0 & \text{if } l_i^{(p)} = \xi^- \end{cases} \tag{14}$$

where $netl_i^{(p)}$ is the net input of the i-th output unit representing the i-th bit of the label when the p-th training pattern is presented to the encoder, $l_i^{(p)}$ the corresponding target value and

$$\xi^+(\xi^-) = \begin{cases} 1\ (0) & \text{if } f_i(x) = \frac{1}{1+e^{-x}} \\ & \\ 1\ (-1) & \text{if } f_i(x) = \frac{2}{1+e^{-x}} - 1 \end{cases} \tag{15}$$

where $f_i()$ is the sigmoidal activation function used by the i-th output.

When the LRAAM is able to map the labels correctly, setting the steepnesses of the corresponding output units to a large value corresponds to transform the sigmoids in step functions. Since the labels are mapped correctly, the total error on the label will drop to zero, and a single tolerance can be used. After the increasing of the steepnesses for the label output units, learning can concentrate on the pointers since the internal representations are no more influenced by the error on the labels. The steepnesses of the hidden units are maintained low in order to reduce the probability for the hidden units to saturate.

In order to develop a robust LRAAM, the action described in point 2 can be postponed until the distance between any outputs corresponding to opposite representations (ξ^+ and ξ^-) for each unit representing the label is larger than a given positive value. In this way we avoid the noise in the activation of the hidden layer to turn a correct map of the label into a wrong one.

3.3.1 Examples of encoding of single structures

In this section we discuss two examples of encoding of single structures using the LRAAM model. The first structure under consideration is the graph shown in Figure 6.

The label of each node is represented by a sequence of boxes. Each box represents a bit of information and the last tree boxes are used to encode the void pointer condition: if the box contains a full circle, then the corresponding pointer field is void. This graph was chosen as an example since, even if it is complex enough, it can be encoded by a $16-3-16$ network with symmetrical sigmoids so that a 3D view of the pointers can be given. The training method discussed in the previous section was employed. The following

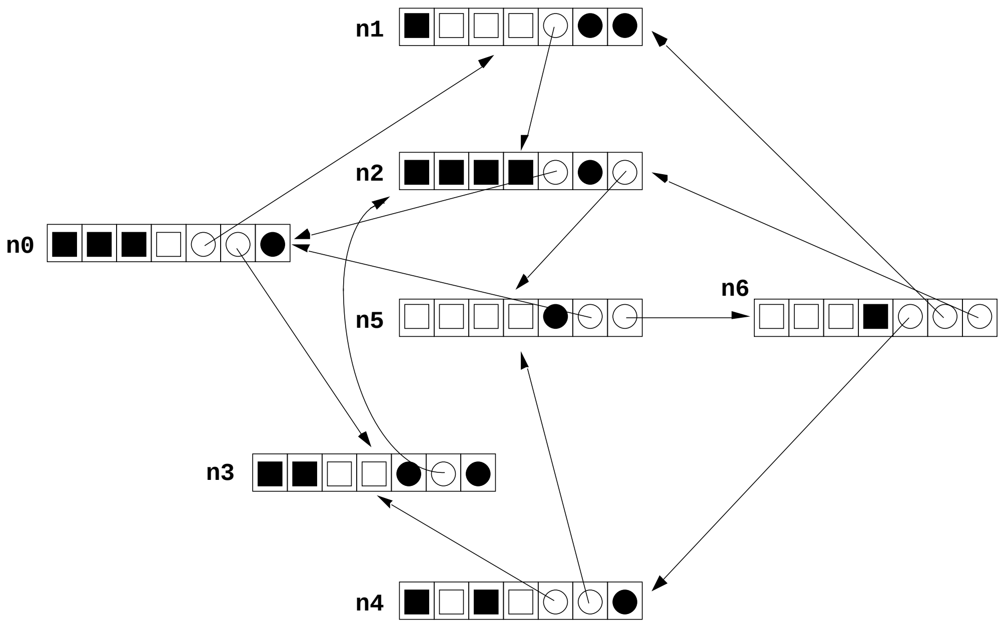

Figure 6: The codified labeled graph.

parameters were used: $\forall h, vo_h = 0.5; \forall l, vo_l = 2.5, VO_l = 6; \forall p, vo_p = 1; \eta = 0.07$(learning rate);$\mu = 0.3$(momentum);$\nu = 0.05$. The weights were updated by epoch. Actually, the action described in point 2 of the training method was postponed until the modulus of the output of the label units was above 0.2. This was made to render the encoder less sensitive to noise in the encoded patterns. The void pointers were initially set randomly. The training employed 5450 epochs to reach the stop criterion. Other simulations showed both faster and slower training times, and in some cases no learning at all since one of the patterns maintained a high error. The decoded representations of the nodes obtained following the paths $(n_o, n_1, n_2, n_5, n_6, n_4)$ and (n_0, n_3) are shown in Figure 7.

Negative activations are represented by white squares and positive activations by black squares. The magnitude of the activation is proportional to the dimension of the square. Note how the void pointers obtain different representations. Since a node can be reached using different paths in the graph, it is interesting to study how a pointer is transformed while running through a cycle more than once. The cycles (n_o, n_1, n_2, n_0) and $(n_o, n_1, n_2, n_5, n_6, n_4, n_3, n_2)$ were tested. The results of the first test are reported in Figure 8.

Each pointer is represented as a point in the bipolar cube. The decoding of a pointer p_i into another pointer p_{i+1} is represented by an arrow starting from the point representing p_i and arriving in the point representing p_{i+1}. The decoding started from the pointer p_0 and, after few cycles, the pointers converged to representations able to decode correctly the associated labels and pointers. The same results were obtained for the other test (see Figure 9).

Thus the encoding obtained can be considered robust, since the information contained in the pointers do not deteriorate with the use. An interesting aspect revealed by these tests is that the final stabilized representation for a pointer depends on the context, i.e., on which cycle is traversed.

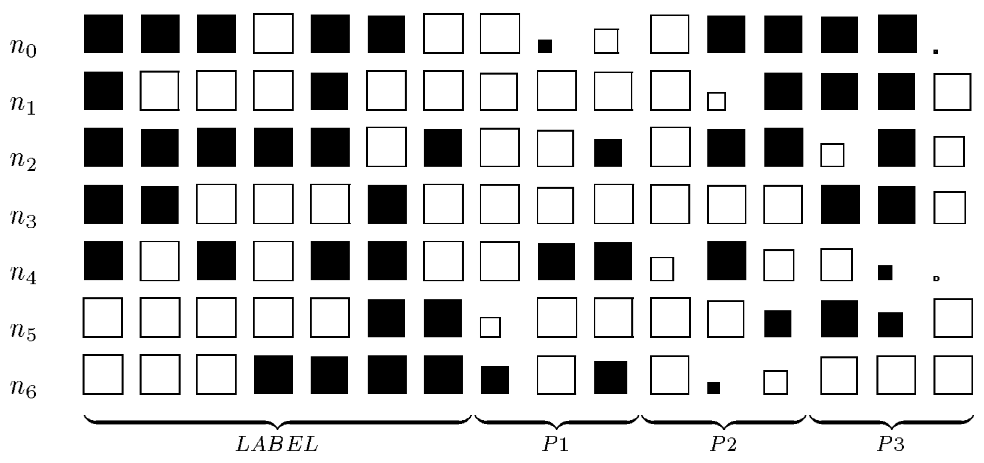

Figure 7: Distributed representations of the graph's nodes.

For example, the pointer p_2 has two slight different representations according to the cycle in which the pointer is considered: in the cycle (n_o, n_1, n_2, n_0) its stable representation is $(-0.926, -0.993, -0.966)$, whilst in the cycle $(n_0, n_1, n_2, n_5, n_6, n_4, n_3, n_2)$ it is $(-0.980, -0.967, -0.965)$.

Since the equivalent representations of the pointers are very similar, the whole graph can be represented in the bipolar cube by choosing for each pointer one of its representations, as shown in Figure 10.

The second structure we have encoded and which we will use in the next section, is the tree shown in Figure 11. A 18-6-18 network and the following learning parameters were used: $\forall h, vo_h = 0.5; \forall l, vo_l = 2.5, VO_l = 6; \forall p, vo_p = 1; \eta = 0.05; \mu = 0.1; \nu = 0.05$. The training stopped after 1719 epochs. The decodified representations of the pointers obtained by the pointer to the root of the tree are shown in Figure 12. Even in this case the NIL pointer is represented by different patterns of activation (see Figure 13).

It is worth to note that only two representations of the NIL pointer were very close, namely NIL_{10} and NIL_{16}. The result of a cluster analysis of the distributed representations is shown in Figure 14. It can be noted that the representations for the void pointer are located mainly on the bottom of the cluster tree, meanwhile the representations for non void pointers are located mainly at the top. The distances among the nearest clusters are reported in table I.

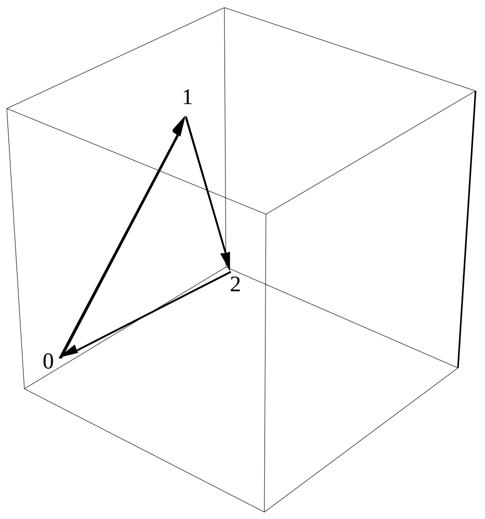

Figure 8: Recursive decoding of the cycle (n_o, n_1, n_2, n_0).

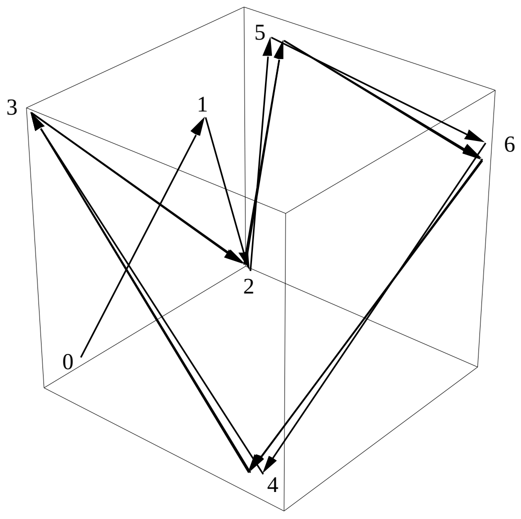

Figure 9: Recursive decoding of the cycle $(n_o, n_1, n_2, n_5, n_6, n_4, n_3, n_2)$.

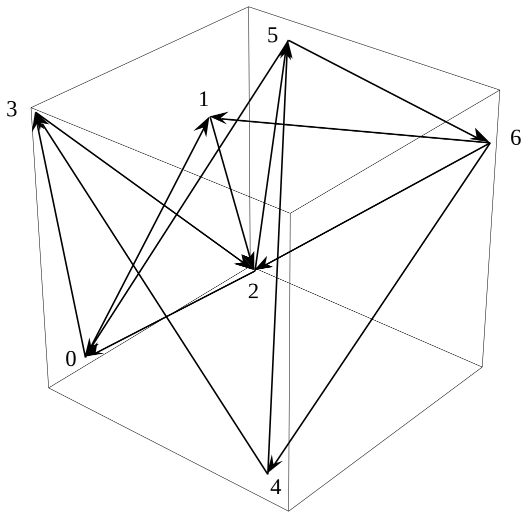

Figure 10: Distributed representations of the pointers of the graph.

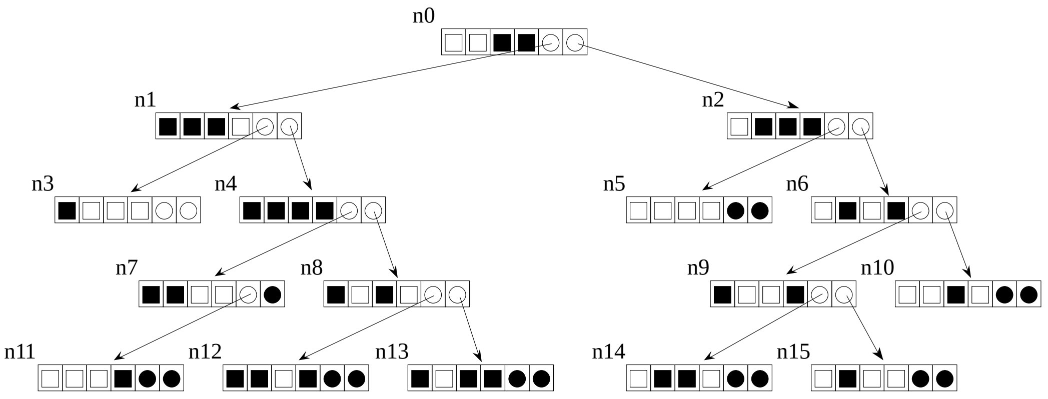

Figure 11: The encoded tree.

$\|\vec{c_1} - \vec{c_2}\|$	$\vec{c_1}$	$\vec{c_2}$
0.152643	(nil_{16})	(nil_{10})
0.260000	(nil_{15})	(p_{11})
0.303315	(nil_{13})	(p_5)
0.419076	$(nil_{10}\ nil_{16})$	(nil_1)
0.440908	(nil_7)	(nil_2)
0.500500	(nil_{11})	(p_4)
0.633973	$(nil_1\ nil_{10}\ nil_{16})$	(nil_4)

Table I: Minimum distances among clusters.

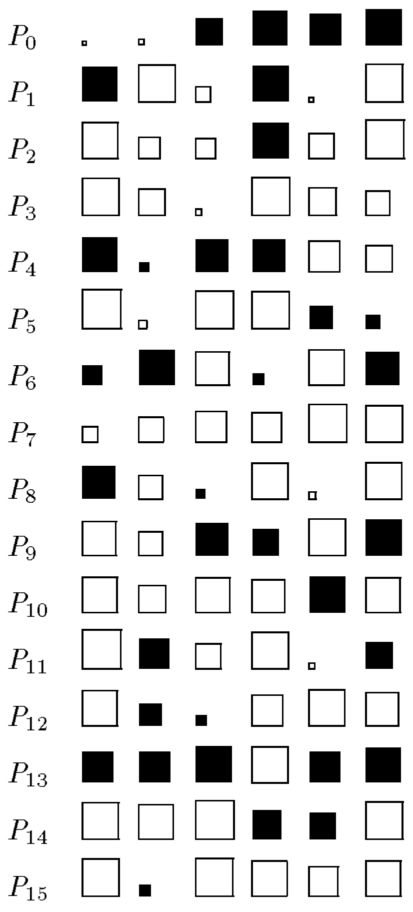

Figure 12: Distributed representations of the pointers of the tree.

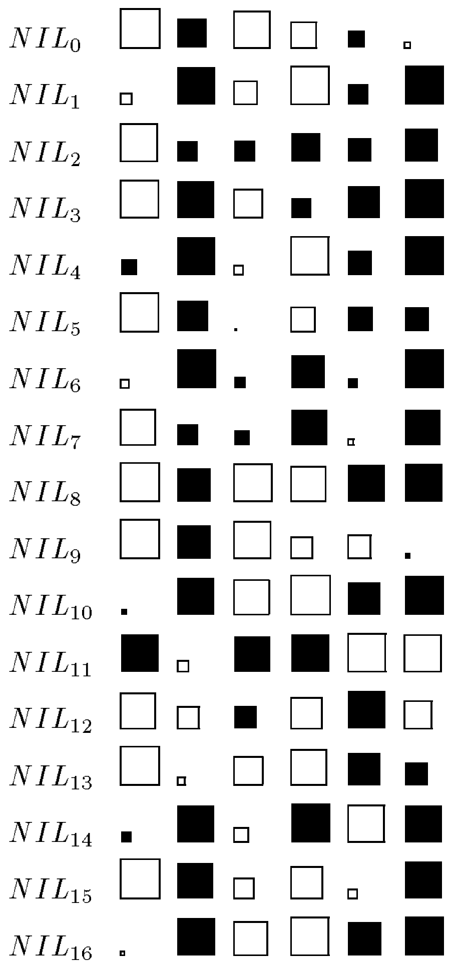

Figure 13: Representations obtained for the null pointer.

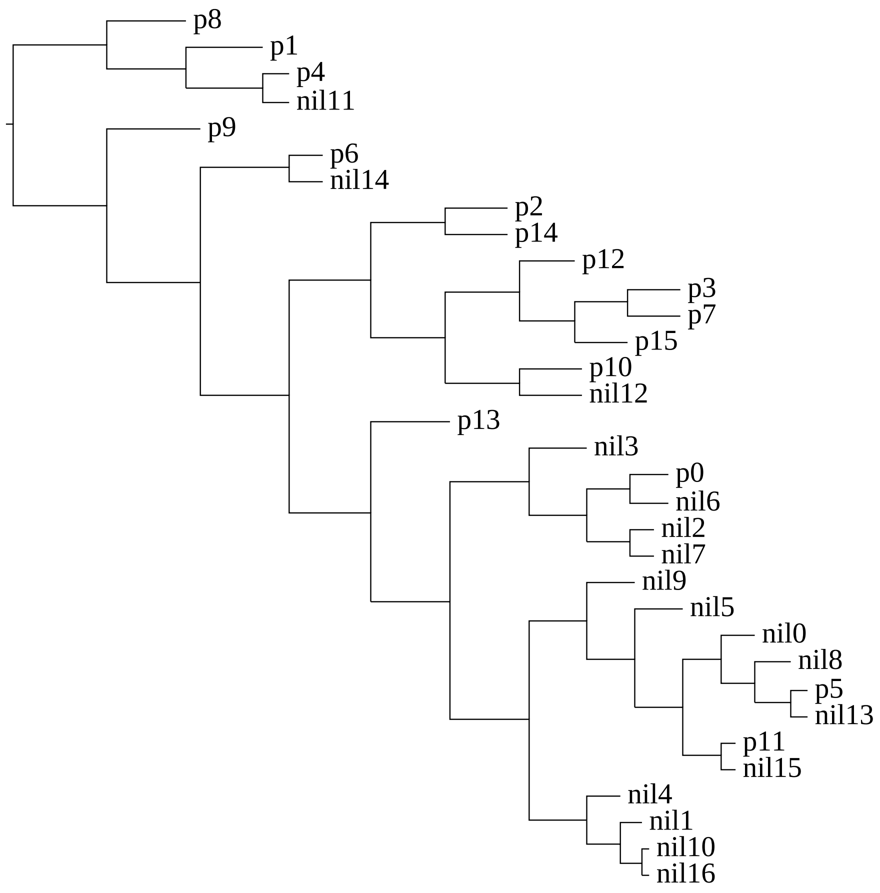

Figure 14: Cluster analysis of the distributed representations obtained for the tree.

4 Retrieval of Information in LRAAMs

Retrieval of coded information is performed in RAAMs through the pointers. All the terminals and nonterminals can be retrieved recursively by the pointers to the whole tree encoded in a RAAM. If direct access to a component of the tree is required, the pointer to the component must be stored and used on demand.

In an LRAAM it is possible to access a component of the encoded structure in other ways if the Encoder Network is transformed in a Bidirectional Associative Memory (BAM) [Kos92].

A BAM consists of two layers of processing elements, name them layer B_H and B_O, that are fully interconnected between layers with weight matrices M_h, from B_H to B_O, and M_o, from B_O to B_h. The weights matrices are such that $M_o = M_h^T$, where M_h^T is the transpose matrix of M_h. The units in a BAM may, or may not, have feedback connections to themselves. In this section we will deal only with BAMs without feedback connections to themselves. We will indicate with x_i a unit in B_H and with y_i a unit in B_O. The output of x_i and y_i in a *bivalent BAM* is defined as:

$$y_i(t+1) = \begin{cases} +1 & net_i^y > 0 \\ y_i(t) & net_i^y = 0 \\ -1 & net_i^y < 0 \end{cases}$$

$$x_i(t+1) = \begin{cases} +1 & net_i^x > 0 \\ y_i(t) & net_i^x = 0 \\ -1 & net_i^x < 0 \end{cases}$$

where $net_i^x = \sum_{j=1}^{m} B_{O_{ij}} y_j$, $net_i^y = \sum_{j=1}^{n} B_{H_{ij}} x_j$, m is the number of units in B_O and n is the number of units in B_H.

To retrieve information using a BAM, the following steps are performed:

1. Apply an initial vector pair, $(\vec{x}_0, \vec{y}_0)$, to the units of the BAM.

2. Update the output values of the B_O layer.

3. Update the output values of the B_H layer.

4. Repeat steps 2 and 3 until there is no further change in the units on each layer.

If a stable state $(\vec{x}_{stable}, \vec{y}_{stable})$ is reached, the retrieval procedure stops. The stable state represents the retrieved memory. Note that steps 2 and 3 can be exchanged, according to which part ($\vec{x}_0$ or $\vec{y}_0$) of the initial vector pair is considered relevant to retrieve the desired memory.

In the next subsection we show how to transform the Encoder Network of an LRAAM in a particular instance of a BAM and discuss which kind of operations we need to perform on it. Some access procedures using the BAM are then defined in Subsection 4.2.

4.1 Transforming the Encoder Network into a BAM

The Encoder Network can be considered as a BAM. In fact, it suffices to feed back its output into its input units to obtain a BAM. Thus, given an Encoder Network $(E_I, E_H, E_O, W_h, W_o)$, where E_I is the set of input units, E_H the set of hidden units, E_O the set of output units, W_h the weight matrix from E_I to E_H and W_o the weight matrix from E_H to E_O, we can define the BAM (B_H, B_O, M_h, M_o), where $B_H = E_H$, $B_O = E_O$, $M_o = W_o$ is the weight matrix from B_H to B_O and $M_h = W_h$ is the weight matrix from B_O to B_H.[3] Actually, the obtained BAM is not standard, since in general $M_h \neq M_o^T$, even if they often are similar. Moreover, the update rule for the units is different with respect to the standard one, since it uses a sigmoidal activation function. To give an idea of how much M_h may be similar to M_o^T in practice, in Figure 15 we have reported the weight matrix of the *LRAAM* encoding the graph shown in Figure 6 and in Figure 16 the weight matrix of the *LRAAM* encoding the tree shown in Figure 11.

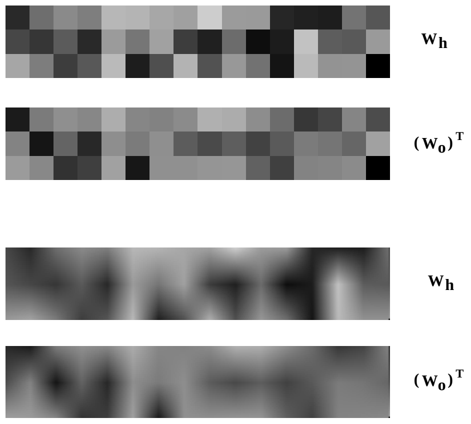

Figure 15: Visual representation of the weight matrix of the *LRAAM* (16-3-16) encoding the graph shown in Figure 6.

The biases and the steepnesses are not represented. On the top of each picture there are two strips representing the weight matrices where each weight is represented by a box whose grey level represents the magnitude of the weight. Light boxes represent negative weights, dark boxes positive weights. Since with this kind of representation similarities between the two matrices are not clear at first glance, an interpolated version of them is given in the second half of the picture. With this representation it is readily clear that the matrices W_h and W_o^t have several common features.

Note that, if the training of the Encoder Network leads to perfect learning, then all the vector pairs $(\vec{\imath}_k, \vec{p}_k)$, where $\vec{\imath}_k$ is the k-th input to the LRAAM and $\vec{p}_k$ the corresponding pattern of activation of the hidden units (pointer), are stable states of the BAM. In practice we will see in the next section that, since the learning is not perfect, the pairs $(\vec{\imath}_k, \vec{p}_k)$ may be either very close to stable states or very different from them.

In order to extract information from a BAM, we need to define a class of modes of operation for it. Given the BAM (B_H, B_O, M_h, M_o), we define a *mode of operation* as a

[3]This is possible because, in an encoder, each input node has a corresponding output node.

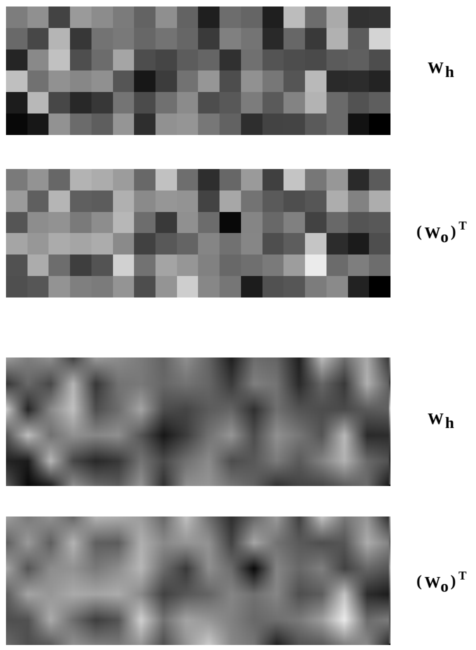

Figure 16: Visual representation of the weight matrix of the *LRAAM* (18-6-18) encoding the tree shown in Figure 11.

subset FIX of units of it. We impose that $FIX \subseteq B_H$ or $FIX \subseteq B_O$. The idea is that the initial value of the units in FIX is maintained fixed during the execution of the recall procedure, i.e., the units in FIX are not updated. Moreover, we allow that only units on the same layer can be in FIX. Our convention will be that the first updating of the units will involve the layer which does't have units in FIX. Note that, $FIX = \emptyset$ leads to the standard recall procedure for a BAM.

This definition of mode of operation, gives us a way to describe in a compact way how to use a BAM to retrieve information. In the next subsection we show how different FIX sets may be used to retrieve different types of information from an LRAAM.

4.2 Access procedures

In the previous section we have shown how the Encoder Network of an LRAAM can be used to define a BAM and noted that, under the hypothesis of perfect learning for the LRAAM, the pairs $(\vec{\imath}_k, \vec{p}_k)$ are stable states (memories) for the BAM. Moreover, we have defined a class of modes of operation for the BAM. In this section we introduce the concept of *access procedure* for an LRAAM, as a triple $(FIX, \vec{z}_{fix}, \vec{z}_{nofix})$, which specifies the mode of operation of the BAM (FIX), a vector of initial values $\vec{z}_{fix}$ for the units in FIX, and a vector of initial values $\vec{z}_{nofix}$ for the units not in FIX. An *execution* of the procedure $(FIX, \vec{z}_{fix}, \vec{z}_{nofix})$ is obtained by instantiating $\vec{z}_{fix}$ and $\vec{z}_{nofix}$. The execution *converges* if $\exists t$ such that $\vec{z}_{nofix}(t) = \vec{z}_{nofix}(t+1)$, and $\vec{res} = \vec{z}_{nofix}(t)$ is the *result of the execution.* Two particular modes of operation lead to the *decoding* and *encoding* procedures:

- $(B_H, \vec{p}_k, \vec{*})$ decodes the pointer $\vec{p}_k$;

- $(B_O, [\vec{l}_k, \vec{p1}_k, \cdots, \vec{pr}_k], \vec{*})$ encodes the component $[\vec{l}_k, \vec{p1}_k, \cdots, \vec{pr}_k]$;

where $\vec{*}$ is any real valued vector, usually chosen with random components, and $[\vec{l}_k, \vec{p1}_k, \cdots, \vec{pr}_k]$ is a vector composed by the subvectors $\vec{l}_k, \vec{p1}_k, \cdots, \vec{pr}_k$. The encoding and decoding procedures are the same as defined and used in both RAAMs and SRAAMs. The general structure of the LRAAM allows us to define other interesting procedures. In order to do that, we have to recall that the layer B_O is composed by a label field, L, and one or more pointer fields, $P_1, \cdots, P_r$.

The first useful procedure that we can define is the direct access to a component of the encoded structure by using the label as key:

$$(L, \vec{l}_k, \vec{*})$$

This procedure is not as reliable as the decoding and encoding procedures. In fact, also if the learning converges, it is not guaranteed that the result of the computation, $\vec{res}$, is the expected one. In several cases, however, it is possible to know if $\vec{res}$ is a wrong result. In fact, $\vec{res}$ contains not only the information on the pointer fields of the component, but also the pointer $\vec{p}_k$ to the component itself. Thus, if the label $\hat{\vec{l}}_k$, obtained by the procedure $(B_H, \vec{p}_k, \vec{*})$, is different from $\vec{l}_k$, it means that $\vec{res}$ is not the correct result. In the next section we will see that nothing about the correctness of $\vec{res}$ can be said if $\hat{\vec{l}}_k = \vec{l}_k$, since in some cases the information regarding the pointers may be incorrect even if the label information is correct. Obviously, the more the label covers the B_O layer, the more the procedure will be reliable.

The dual procedure of the direct access by label, is the retrieval of pointer $\vec{p}_k$ and label $\vec{l}_k$ given the pointers $\vec{p1}_k, \cdots, \vec{pr}_k$:

$$(P_1 \cup P_2 \cup \cdots \cup P_r, [\vec{p1}_k, \cdots, \vec{pr}_k], \vec{*})$$

This procedure is very useful in trees processing, since it allows to retrieve all the information of a node by knowing its children. In this case, the test on the correctness of $\vec{res}$ must be performed between $[\hat{\vec{p1}}_k, \cdots, \hat{\vec{pr}}_k]$ and $[\vec{p1}_k, \cdots, \vec{pr}_k]$, where $[\hat{\vec{p1}}_k, \cdots, \hat{\vec{pr}}_k]$ is again obtained by the procedure $(B_H, \vec{p}_k, \vec{*})$.

Other procedures can be defined by considering FIX as the union of any combination of the sets $L, P_1, \cdots, P_r$.

4.2.1 Examples of retrieval

We will use the *LRAAMs* developed in Section 3.3 to show the efficiency of our access procedures. The *LRAAMs* codifying the graph and the tree were transformed in *BAMs*, as discussed in Section 4.1. The results obtained by applying the access procedure by label to them are reported in Table II. The vector $\vec{*}$ was initialized with random real numbers in the range $[-1, 1]$ and 100 trials for each label were performed. The labels $\hat{\vec{l}}_k$ and $\vec{l}_k$ were considered different ($\hat{\vec{l}}_k \neq \vec{l}_k$) when at least one component of the vectors differed for more than 0.1 (0.06 for the access procedure by pointers).

The first column in the table reports the label used in the procedure, the second column the *success rate* obtained, where a success is a correct retrieval of the information in the

ACCESS BY LABEL				
GRAPH				
label	success rate	wrong rate	error rate	mean
l_0	100%	0%	0%	7.35
l_1	100%	0%	0%	36.05
l_2	100%	0%	0%	6.04
l_3	100%	0%	0%	3.99
l_4	100%	0%	0%	23.12
l_5	100%	0%	0%	18.12
l_6	100%	0%	0%	29.26
TREE				
label	success rate	wrong rate	error rate	mean
l_0	0%	100%	0%	16.48
l_1	94%	6%	0%	14.57
l_2	47%	53%	0%	16.92
l_3	100%	0%	0%	18.07
l_4	97%	0%	3%	32.64
l_5	100%	0%	0%	16.03
l_6	49%	51%	0%	27.50
l_7	42%	58%	0%	27.10
l_8	57%	43%	0%	62.45
l_9	20%	0%	80%	14.75
l_{10}	100%	0%	0%	19.11
l_{11}	100%	0%	0%	10.83
l_{12}	100%	0%	0%	19.12
l_{13}	29%	71%	0%	23.87
l_{14}	100%	0%	0%	12.09
l_{15}	100%	0%	0%	13.11

Table II: Results obtained accessing the data by label.

node, the third column the rate of trials which led to a *wrong* retrieval ($\vec{\tilde{l}}_k \neq \vec{l}_k$), the fourth column the rate of *errors* ($\vec{\tilde{l}}_k = \vec{l}_k$ and incorrect retrieval), the last column the mean number of iterations employed by the BAM to stop. Note that a wrong retrieval is not an error, since we know that the procedure failed in retrieving the expected information and we can run again the procedure, until the condition $\vec{\tilde{l}}_k = \vec{l}_k$ is satisfied. However, a high rate of wrong retrievals reduces the efficiency of the procedure, since several attempts must be performed to obtain a correct answer. The efficiency of a procedure depends also on the mean number of iterations employed by the BAM to stop (last column in the table).

It can be seen from Table II that the procedure worked very well for the graph (no errors and no wrong retrievals), but had some problems with the tree. In particular, the procedure applied to the tree showed a dependence on the initial random vector $\vec{*}$. A range from one to three different results for the same label was observed. The pointers to the nodes retrieved by the labels are shown in Figure 17. The retrieved pointers which satisfied the condition $\vec{\tilde{l}}_k = \vec{l}_k$ are shown on the left side of the picture ($success$ + $error$), the others on the right side ($wrong$). The pointers on the left side which led to errors are marked at the end by an asterisk (*).

The procedure was very efficient when called with labels belonging to leaves (with the exception of the label l_{13}), but was unable to recover the information of the root of the tree. Moreover, it had a high rate of errors (80%) for the label l_9. This disparity of performance between terminal and nonterminal nodes may be due to the fact that, during the training of the encoder network, patterns representing leaves reach stable representations sooner than others because of the NIL pointers, whose output representations are reused in the training set. It is as if the memories of the BAM are constructed around those stable representations.

No errors, instead, were made by the access procedure using children pointers [4] when applied to nonterminal nodes of the tree (see Table III).

The pointers to the children of a nonterminal node were used as fixed activations, with the exception of node n_7 for which only one pointer was used. The success rate obtained with the access by pointers was lower than the one obtained with the access by label, and the number of results for the same pointers ranged from three to eight.

The reduction of the error rate may be attributed to the fact that the steepness of sigmoids of units representing the label is higher than the steepness of units representing pointers, and consequently it is more probable that the same label could be obtained by different hidden activations (pointers), as happened for labels l_4 and l_9.

Another difference with respect to the access by label is the lower mean number of iterations employed by the BAM. This is due to the bigger dimension of the FIX set which simplifies the dynamics of the BAM. This behaviour matches the general claim that relaxation in a connectionist system is more efficient when more constraints are imposed. Note that a classical symbolic system becomes less efficient when more constraints are imposed.

In conclusion, the $LRAAM$ model does not only extend the class of structures which can be encoded by a neural network, but it also allows the definition of access procedures different from the classical access by pointer.

[4]The children pointers of a node are used as key in the access procedure.

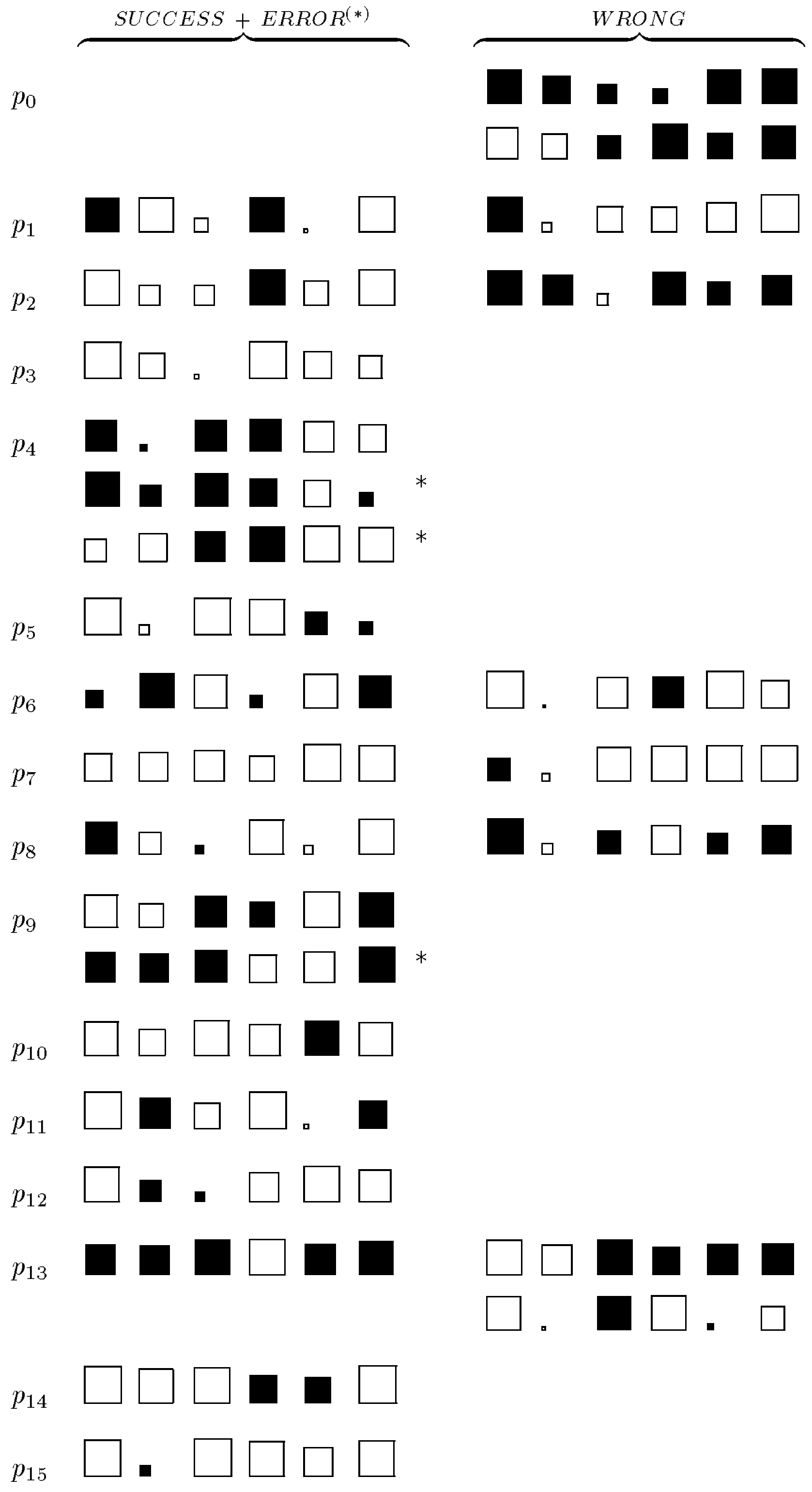

Figure 17: Pointers of the tree retrieved by label.

ACCESS BY CHILDREN POINTERS				
TREE				
node	success rate	wrong rate	error rate	mean
n_0	49%	51%	0%	6.29
n_1	10%	90%	0%	8.55
n_2	40%	60%	0%	12.48
n_4	78%	22%	0%	6.57
n_6	9%	91%	0%	6.22
$n_7^{(*)}$	14%	86%	0%	14.01
n_8	14%	86%	0%	7.87
n_9	28%	72%	0%	6.07
(*) one pointer				

Table III: Results obtained accessing the data by children pointers.

5 Encoding of a Set of Structures and Generalization

The standard generalization test proposed by Pollack for the RAAM cannot be performed directly on the LRAAM model due to the presence of NIL pointers which cannot be guessed when trying to compose a leaf with a well-formed structure. A solution to this problem would be to use the BAM in order to find a fixed point for the NIL pointers: it suffices to access the BAM using the label on the leaf as key; if a fixed point is found, then the representations obtained for the NIL pointers can be used in the representation of the leaf. This process, however, is too strict with respect to the standard test since it requires an asymptotically stable representation for the leaf, which may not happen in a RAAM where the test is not iterated on time. Another solution to this problem can be to perform a gradient descent on the pointers maintaining the label fixed. This can be performed using the inversion technique proposed by Kindermann and Linden [KL90]. Using this method, representations for the pointers which satisfy the generalization test, without the constraint to be an asymptotically stable memory of the BAM, can be found. Obviously, this technique is computationally more expensive since it involves gradient descent.

Both of the above solutions, however, must face the combinatorial explosion of the method when the number of different labels increases. For this reason, we prefer to follow a different approach with respect to generalization, that is to try to extract global information on the classes of structures that for sure cannot be encoded by a particular LRAAM. For example, the first question may be: how much powerful is the decoding function, i.e., given a structure does it exist a reduced representation from which the recursive application of the decoding function can extract all the components of the structure?

The approach we use in performing this exploration is very close to the one used by Blank and al. [BMM92], but adjusted to the LRAAM model and extended in scope. In the next section we will give a geometric interpretation of the binary LRAAM model with respect to the decoding process.

5.1 Geometric interpretation of the decoding process

As a first approximation let us consider an LRAAM with binary labels whose sigmoidal output functions in the label field has been replaced by hard limiters [5]. Under this condition, the function computed by each output node of the label field consists of the splitting of the hidden space by an hyperplane whose normal is the vector of weights entering the unit. Thus the set of labels which can be represented by our LRAAM is given by the set of regions in which the hidden space is partitioned by the hyperplanes of the label field. Moreover, we must also consider the constraints on the hidden output, which is limited to the k-dimensional hypercube, where k is the number of hidden units. Consequently all the regions outside the k-dimensional hypercube must be thrown away.

The situation at this point is that each label which is representable by our LRAAM decoder has a region of the hidden space associated to it and, as long as an hidden representation (pointer) will fall in this region, the decoded label remain the same.

Let us now turn the attention to the pointer fields. They are used to transform the actual pointer to the next pointer, which can be, in the case of a binary tree, the pointer to the left or the right child. This new pointer will fall in one of the label regions and the decoded label will be the one associated with it. Thus, if we get some information about how this transformation works, then we can decide if some class of trees, for example the class of trees with labels all equal to *A* on the path obtained by choosing always the left child, is representable by the LRAAM decoder or not.

In order to understand the transformation of a pointer let us consider the decoding part of an LRAAM with two hidden units (see Figure 18).

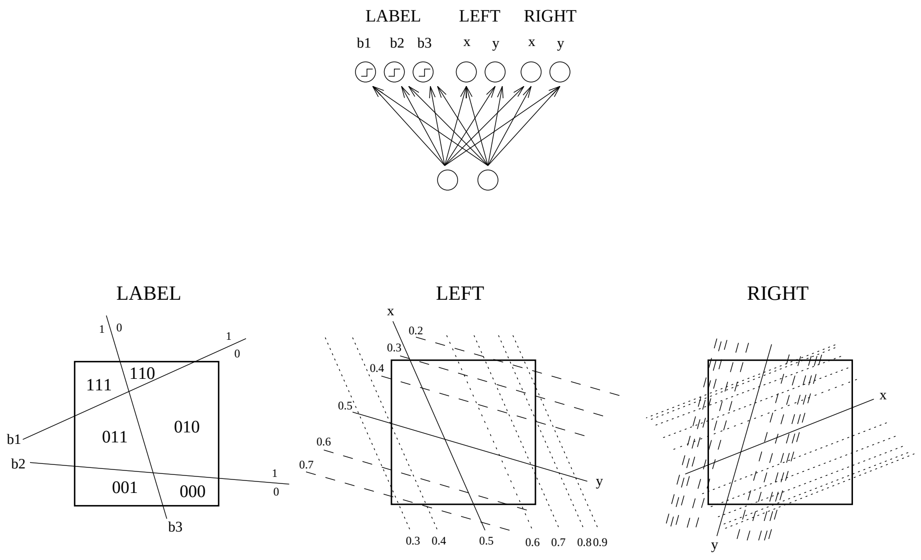

Figure 18: Geometric interpretation of a binary LRAAM.

Consider, for example, the left pointer field of the LRAAM. Each unit of the field splits the xy-plane in two parts. All the points along the split, the line defined by the weight vector, are mapped into 0.5 (or 0, accordingly to the kind of sigmoid used). The other

[5]An hard limiter can be obtained by increasing the steepness of the sigmoid to infinity.

points of the plane are transformed according to equipotential lines parallel to the splitting line and to the modulus of the weight vector. On one side of the splitting line the output value will decrease from 0.5 to 0 (or from 0 to -1) with the increasing of the distance of the point from the splitting line; on the other side of the hyperplane it will increase to 1. The sharpness of the decreasing and increasing is proportional to the modulus of the weight vector.

Actually the transformation implemented by a pointer field is completely described by the analog Hopfield network obtained by feeding back the output of the pointer field to its input. In order to get a feeling of how this transformation works, let us consider two very simple examples where more than a single structure is encoded in a two hidden units LRAAM. The training set for the two examples are shown in Figure 19.

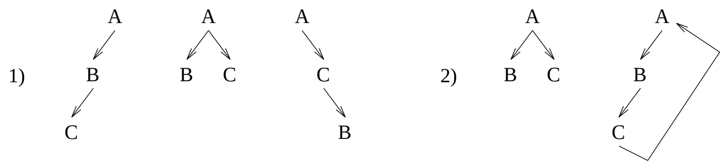

Figure 19: The training sets.

In the first example only three binary trees are present, in the second we have a tree and a graph composed of a single loop. The letters are represented by two bits (A=10, B=01, C=00) and the void conditions for the pointers by another two bits (void pointer=1). The following parameters for the learning procedure were used: $\forall h, vo_h = 0.5; \forall l, vo_l = 2.0, VO_l = 6; \forall p, vo_p = 1; \eta = 0.15; \mu = 0.5; \nu = 0.06$. The first training set was learnt after 4574 epochs, the second one after 2872 epochs. Note that the training sets used for training contained a different instance for each component. Thus, for example, the leaf "C", in the first training set, got two different input patterns: one for the first tree, one for the second. These input patterns are initially different, because of the multiple representations for the void pointer. Actually, after learning, the different representations become almost identical. The resulting partition of the unit square for the first training set is shown in Figure 20. It can be noted that two labels not present in the training set are represented as well: the leaf "A" and the internal node labeled "C" with a left child. In the partition there is also an undefined region since no meaning was assigned to the configuration 11 for the first two bits of the label.

A feeling of how a pointer is transformed by the left and right fields can be gained by looking at how the boundary of the unit square is transformed (see Fig. 21). Since the sigmoidal functions are monotone, the restriction of the transformation to the boundary of the unit square returns a quite synthetic and realistic view of its global characteristics.

This kind of reduced representation of the transformation, however, fails to capture the extent to which the domain is twisted and deformed. In order to remedy to this limitation, we drew as well the vector field of the transformation computed on a small set of points distributed on a grid into the unit square (Fig. 22). The vector field is represented by plotting the domain point and its transformed result (image point) as a vector with the arrowhead ending at the image point.

In the case of the first training set, both the transformations implemented by the left and right fields collapse the unit square into fixed points very close to the boundary of the

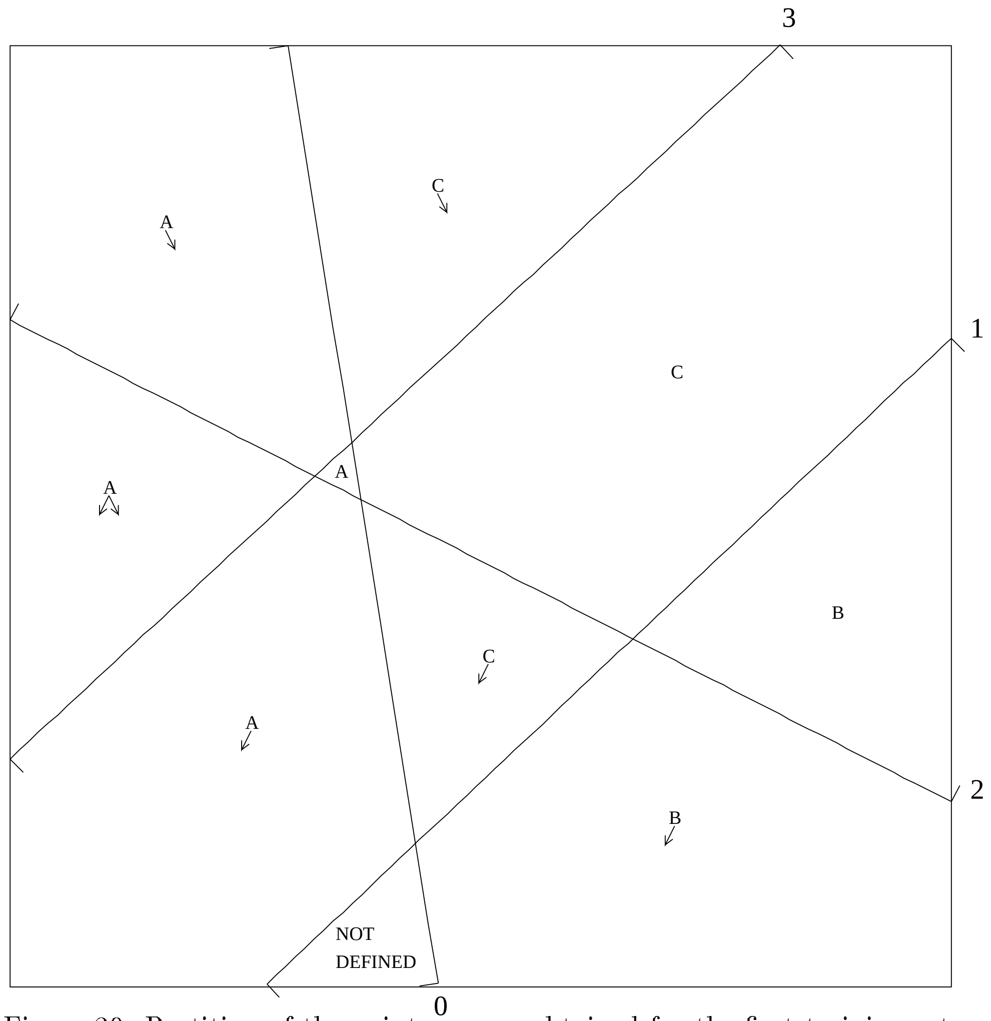

Figure 20: Partition of the pointer space obtained for the first training set.

unit square in a few steps. Both the fixed points are located at the right part of the unit square, where the regions representing leaves laid.

Since the weight matrices entering the pointer fields were invertible, it was possible to draw for each transformation a partition of the unit square were each region is labeled with the label of the image point (see Fig. 23). In fact, called W_l the weight matrix entering the left pointer field and W_r the weight matrix entering the right pointer field, the inverse of the left and right pointer transformations can be written as:

$$(\vec{p})_l^{-1} = W_l^{-1}(\mathbf{f}^{-1}(\vec{p}) - \vec{\Theta}_l), \tag{16}$$

$$(\vec{p})_r^{-1} = W_r^{-1}(\mathbf{f}^{-1}(\vec{p}) - \vec{\Theta}_r), \tag{17}$$

where W^{-1} is the inverse of W, $\vec{\Theta}_l$ and $\vec{\Theta}_r$ are the bias vectors for the left and right pointer fields and $\mathbf{f}^{-1}(\vec{v}) = [f^{-1}(v_1), \cdots, f^{-1}(v_m)]$, i.e., the function computing the inverse of the sigmoidal transformation of the input vector's components. In our case, $f(x) = o_x = \frac{1}{1+exp(-x)}$ and thus $f^{-1}(o_x) = -\log(\frac{1}{o_x} - 1)$.

Now an inverse partition of the unit square can be obtained by applying eq.(16) or eq.(17) to the points laying on the hyperplanes defined by the label field and retaining

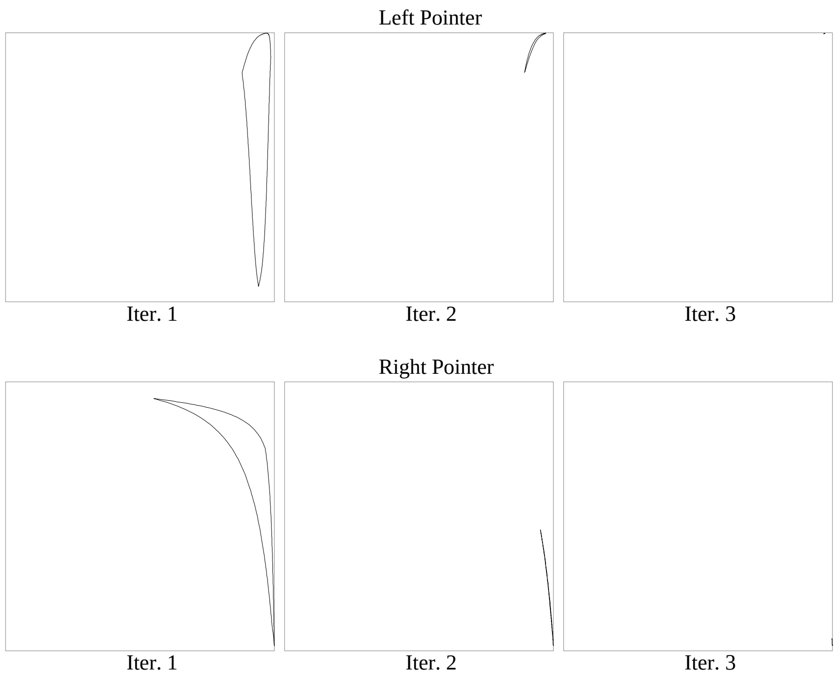

Figure 21: The first three transformations of the unit square boundary by the left and right pointer functions for the first training set.

only the transformed points falling into the unit square. The combination of the original partition of the unit square and the partitions obtained by exploiting the inverse of the left and right pointer transformations allowed us to draw a more complex partition where each region is labeled by structures till to two levels deep (see Fig. 24).

In particular, this partition was obtained overlapping the partitions in Figures 20 and 23, and disregarding the regions not consistent with the correct interpretation of the void pointer bits (the 3th and 4th bit of the label). Partitions representing explicitly deeper structures can be obtained considering the recursive application of the pointer inverse functions to the label hyperplanes and combining the partitions obtained till then. If the LRAAM is able to represent only finite structures (as in our examples) this process will reach a fixed point partition, either because all the transformed hyperplanes are outside the working space, i.e., the unit square minus the regions representing leaves, or because the inverse functions reach a fixed point.

The partition shown in Figure 24 is reminiscent of a fractal set, with the remarkable difference that we are dealing with a finite partition due to the collapsing of each left and right pointer to the respective fixed points of the left and right pointer transformations. In fact, the unit square is subdivided in a top-down fashion, where larger regions are used to represent the different possible roots of a tree, then each root region is partitioned according to the labels of its children and so on recursively, till no more subdivision can occur because of the reaching of the regions representing leaves (where the fixed points of the left and right pointer transformations are located).

An interesting aspect of this kind of representational scheme is that pointers pointing to very different structures get very different representations, while pointers to very similar structures get very similar representations. Moreover, the hierarchical organization of the

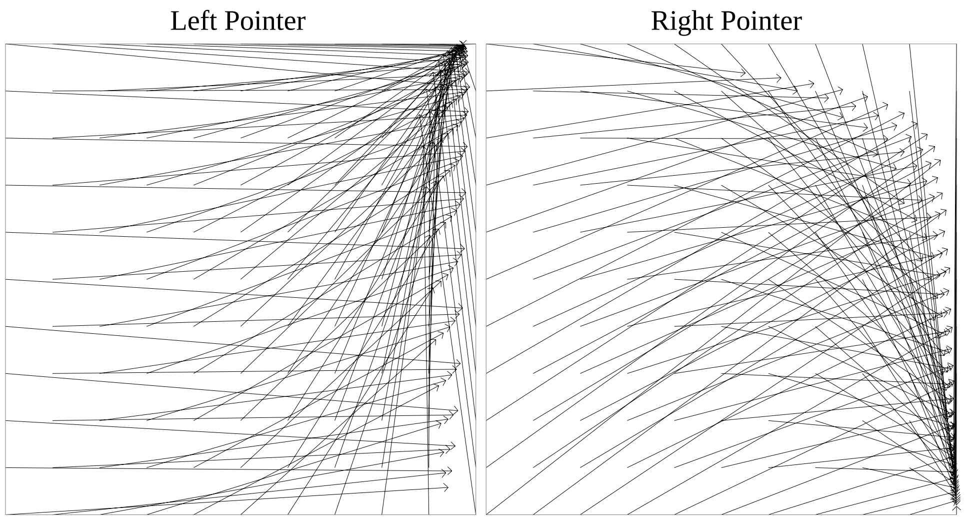

Figure 22: The vector fields of the left and right pointer transformations for the first training set.

compressed representations allows one to consider only a certain number of regions according to the level of discrimination he/she is interested in.

Considering the second training set, in Figures 25-29 we have reported the same kind of plots we presented for the first example.

This time, due to the presence of a loop in one of the structures represented in the training set, the right pointer transformation gets as fixed point a cycle of period two ($\{[0.01, 0.98],[0.984, 0.036]\}$). This cycle is the responsible of the particular form of the plot shown in Figure 26 for the right pointer. Even in this case, the partition of the unit square by the label field is such that labels not present in the training set are represented as well. The compressed representations obtained for the cycle were stable also in this case.

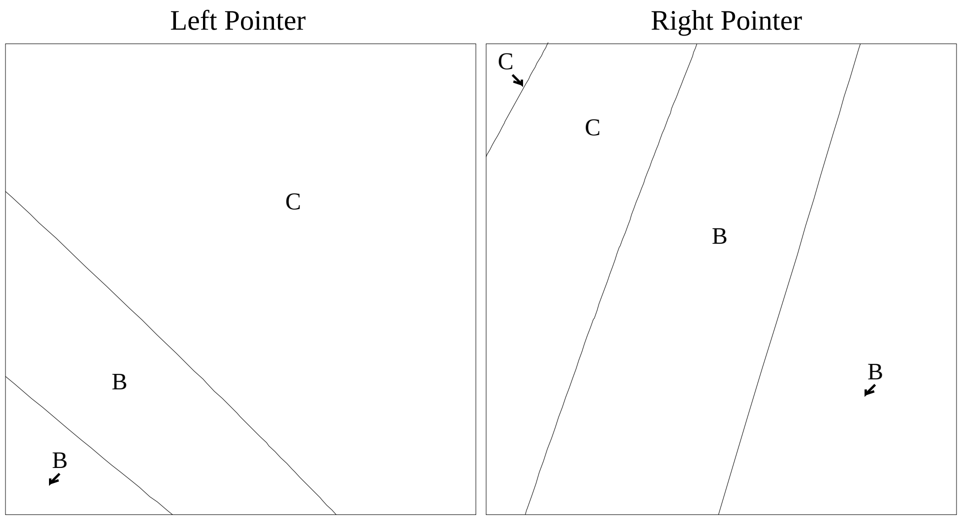

Figure 23: The label partitions after the application of the inverses of the left and right pointer transformations for the first training set.

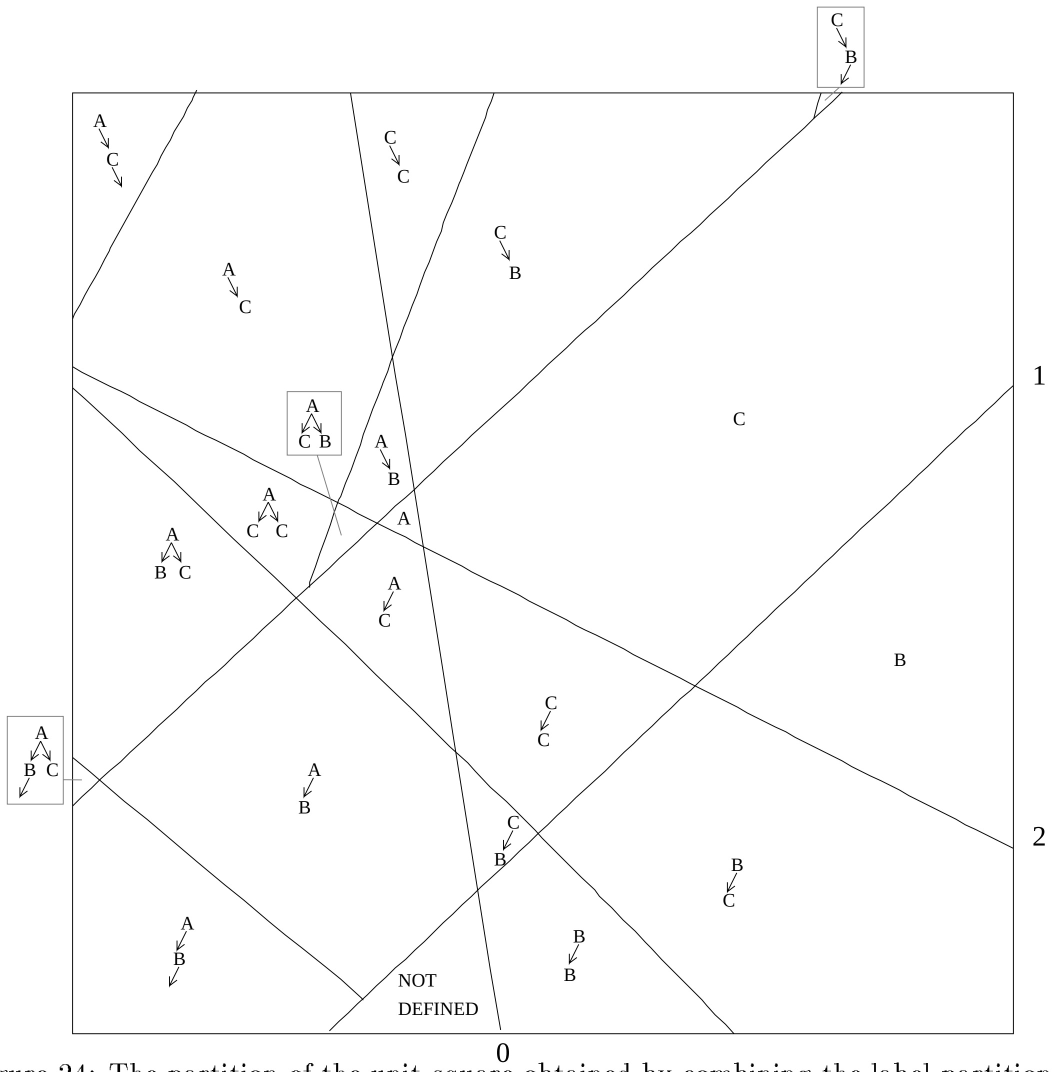

Figure 24: The partition of the unit square obtained by combining the label partition with the partitions after the application of the inverses of the left and right pointer transformations for the first training set.

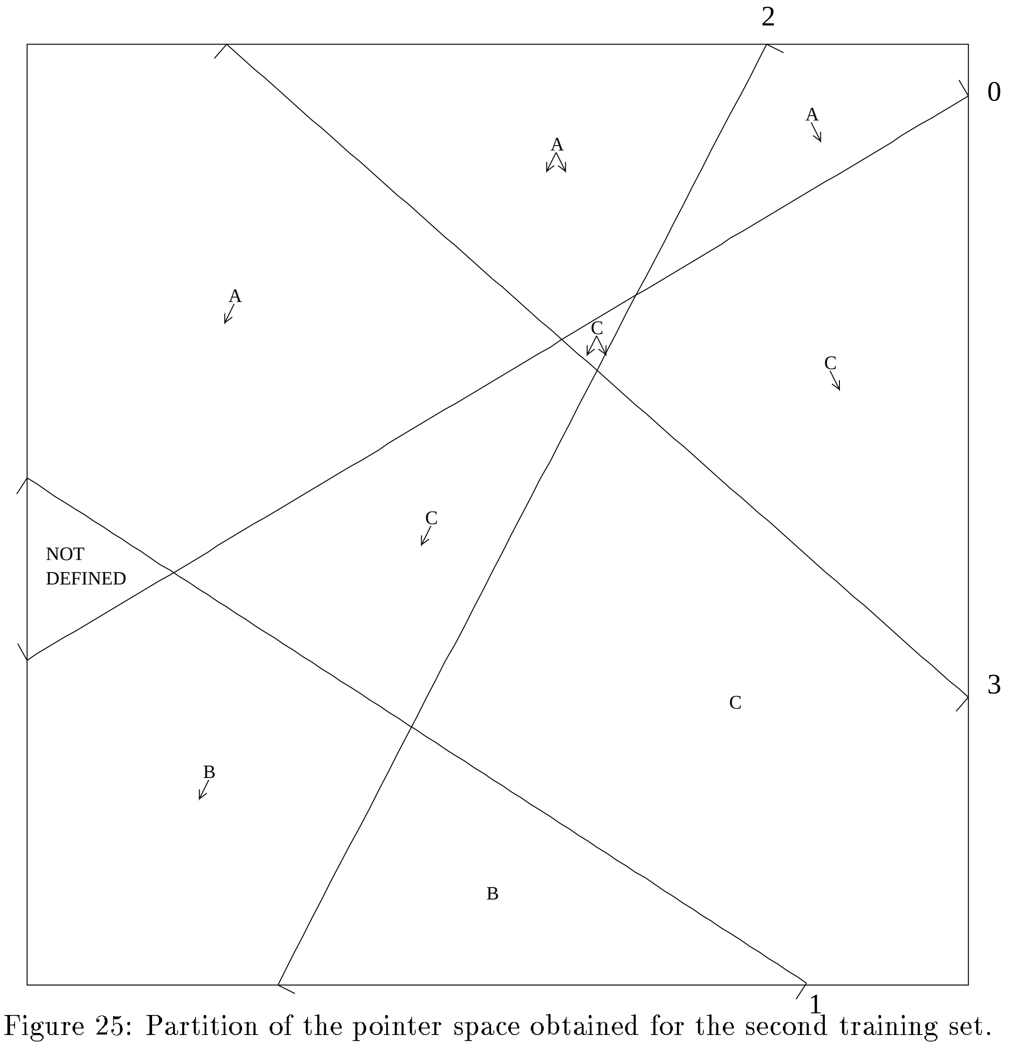

Figure 25: Partition of the pointer space obtained for the second training set.

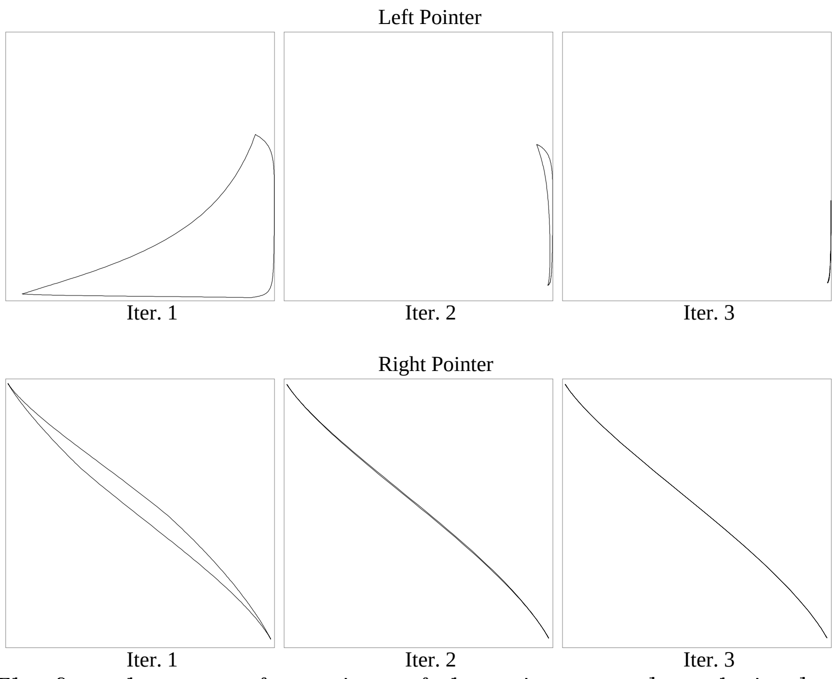

Figure 26: The first three transformations of the unit square boundaries by the left and right pointer functions for the second training set.

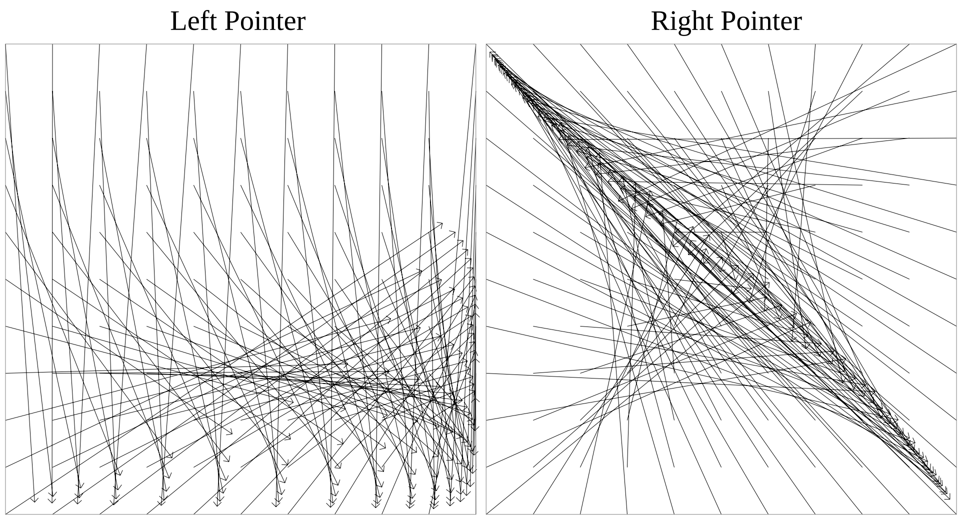

Figure 27: The vector fields of the left and right pointer transformations for the second training set.

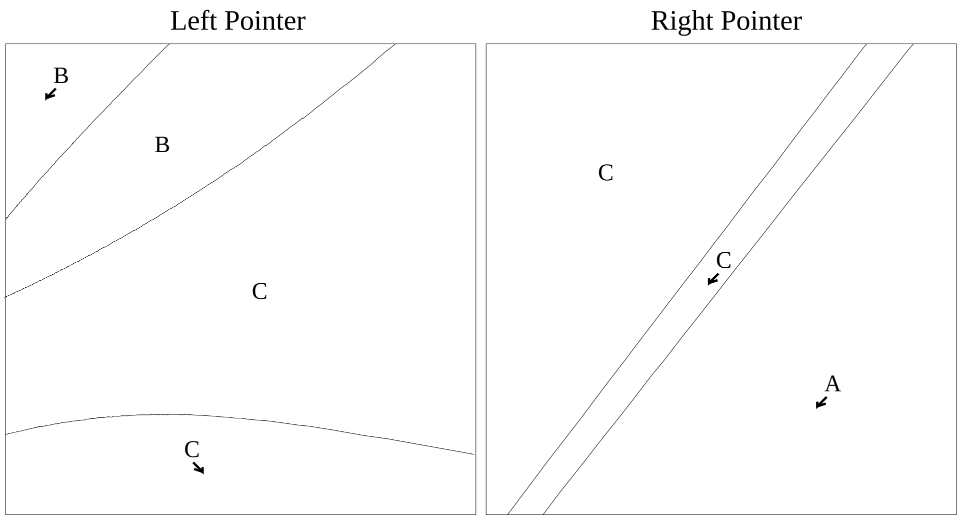

Figure 28: The label partitions after the application of the inverses of the left and right pointer transformations for the second training set.

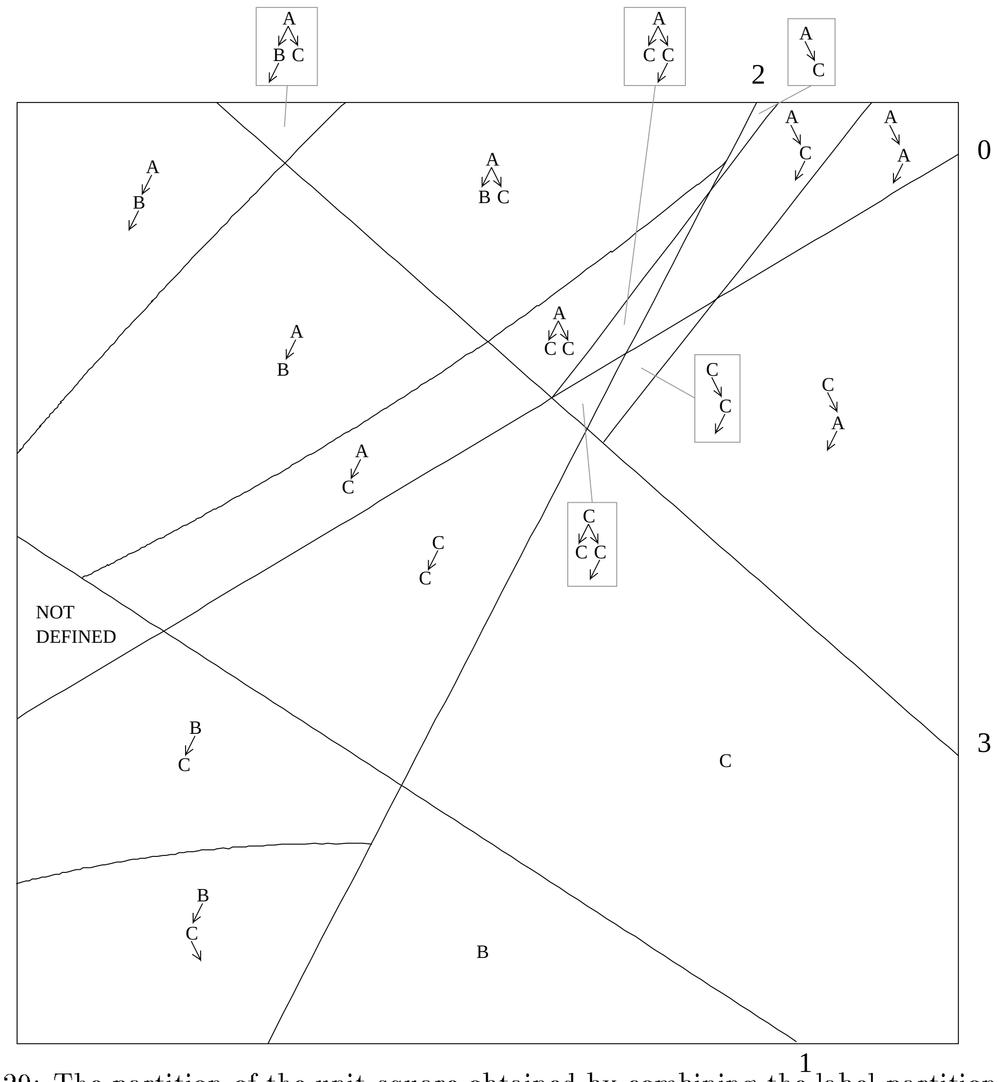

Figure 29: The partition of the unit square obtained by combining the label partition with the partitions after the application of the inverses of the left and right pointer transformations for the second training set.

5.2 The fusion, the missing, and the dragging effects

Three main effects on the representational abilities of the decoder are induced by the underlaying Hopfield-like dynamics: the *fusion*, the *missing*, and the *dragging* effects. The fusion effect means that if in the training set there are two structures with an identical sequence of labels, then with high likelihood the fusion of the structures on the identical sequence belongs to the image of the decoder. Examples of the fusion effect are present both in the first and second learning tasks we have discussed. Two examples of the fusion effect, one for each training set, are reported in Figure 30.

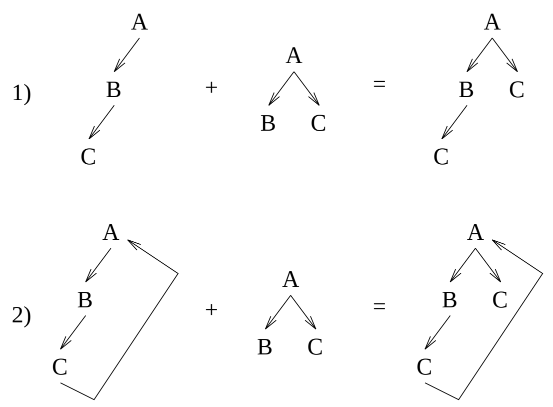

Figure 30: The fusion effect: two examples.

Strictly correlated with the fusion effect is the missing effect. The missing effect means that, with high likelihood, copies with missing components of a structure in the training set are in the image of the decoding as well. These two effects can be understood as an indirect consequence of the proximity of the representations of the labels in the partitions of the pointer space.

Another effect, more correlated with the Hopfield-like dynamics, is the dragging effect. It is present when the vector field of the left and/or right pointer transformations is not very strong. In these cases, the transformed pointers tend to move slowly, i.e., to drag themselves, in a same restricted area which usually contains the regions representing variants of the same label (i.e., with two children, with one child, with no children).

Our opinion is that a large percentage of the generalization ability of the LRAAM is based on these three effects. Obviously it is not possible to foresee a priori how much and where they will appear. However, they should be kept in mind when evaluating the expected generalization of the LRAAM.

5.3 The encoding problem

Till now, we have discussed the representational capabilities of the decoding process in an LRAAM. However, due to the encoding process, not all the points in the pointer space are consistent, i.e., several points returned by the encoding process are not decoded back to the original structure. Moreover, given a trained LRAAM, it is not straightforward how to devise the compressed representation (encoding) of a cycle.

In the previous sections we have seen that the decoder function usually defines a rich representational space starting from few examples in the training set. Therefore, it is worth to try to derive a better encoding procedure, able to retain such a rich structure. In this section we sketch a method to exploit the decoder function in the encoding process.

The first observation is that it is not possible to apply directly the inverse functions defined in eqs.(16-17)[6] in the encoding process because of the multiple representations the leaves, in particular, and each component, in general, get. The problem could be solved by giving a fixed representation to the void pointer. However, this constraint can be very strong since it makes a precise requirement on the shape of the pointers transformations.

Thus, let us assume we have multiple representations for the void pointer. The difficulty relays on the fact that a different representation of the pointer to the same component is needed according to the context in which the component is. Since, in general, it is not plausible to have the whole information about the correct location of the correct representation for the pointer in each possible context, we must assume no particular knowledge on each particular context. The only information we allow is the knowledge of the region in the pointer space where each single label is represented.

The basic idea is to build a network according to the shape of the structure we want to encode using as building blocks the components of the decoding function, i.e., we unroll the decoding function in the space. The compressed representation of the structure is then devised by an inverse gradient descent on the pointer space. Two examples of how to built such networks are shown in Figure 31.

When the structure is free of cycles, the target of the associated net is given by its labels and "don't care" components are introduced dynamically as soon as bits of the labels are mapped on the correct side of the labels' hyperplanes[7]. In this way, the likelihood to get stuck in a local minimum decreases. Moreover, the starting point for the pointer to the structure is chosen according to our knowledge about the region in the pointer space where each single label is represented, so to satisfy at least the first constraint on the label of the root. If we are lucky or our knowledge about the topology of the pointer space is more accurate than the supposed one, we just may choose the correct pointer and no search is needed. In general, however, the chosen pointer will be decoded in a wrong structure and inverse gradient descent is needed.

If the structure contains cycles, then constraints on some pointer fields must be satisfied as well (see Fig. 31). Thus, in general, the error function which must be minimized by the inverse gradient procedure to obtain the correct encoding of a structure is composed by two terms: one which involves labels (and "don't care" symbols), and one which involves pointers.

6 Discussion

The introduction of the LRAAM model can be viewed from different perspectives. It can be considered as an extension of the RAAM model, which allows one to encode not only trees with information on the leaves, but also labeled graphs with cycles. On the other hand it

[6]If they exist.

[7]Or, in order to obtain a more robust encoding, as soon as the input net value is on the correct side and over a given tolerance.

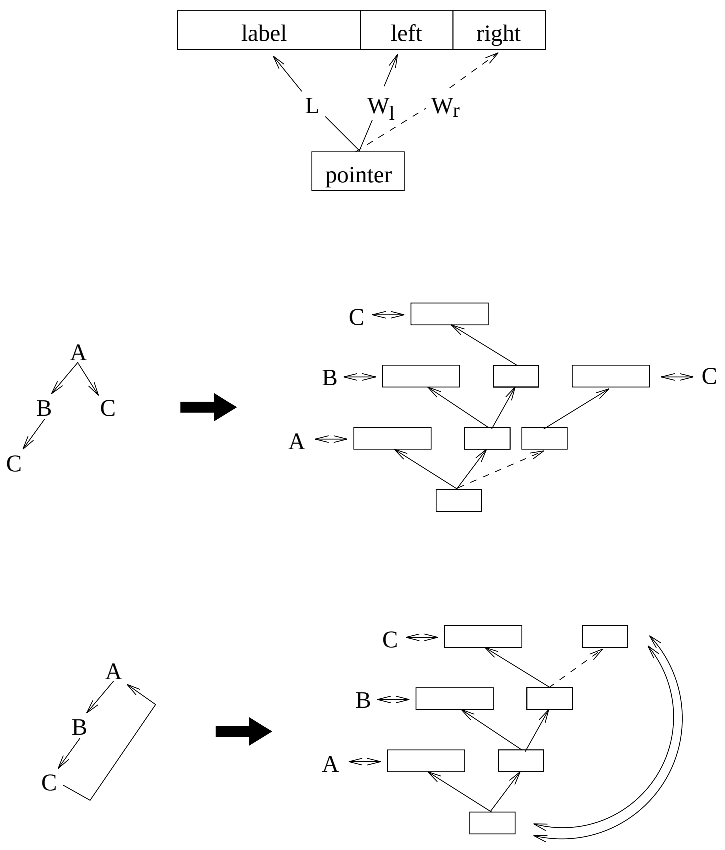

Figure 31: Examples of encoding networks.

can be viewed as an approximate method to build analog BAMs, which actually are analog Hopfield networks with a hidden layer [AAM93].

Most probably an LRAAM is something between them. In fact, while extending the representational capabilities of the RAAM model, it doesn't possess the same synthetic capabilities of the RAAM, since it uses explicitly the concept of pointer. Due to this fact, different subsets of units are used to codify labels and pointers. In the RAAM model, the use of the same set of units to codify labels and reduced representations gives a more natural way to integrate a reduced descriptor previously developed as a component of a new structure. Actually, we guess this ability was the original motivation of the creation of the RAAM model by Pollack since, because of that, it is possible to fill a linguistic role with the reduced descriptor of a complex sentence. In the LRAAM model the same target can be reached, but in a less naturally way. There are two possible solutions. The first one is to store the pointer of some complex sentence (or structure, in general) previously developed in the label of a new structure. This solution, however, calls for a more accurate approximation of the label field, since it would be no more possible to use sharp sigmoids, and for the already discussed constraint between the size of the label field and the pointer one. The other solution would be to have a particular label value which tells us that the information we are looking for can be retrieved using one conventional or particular pointer among the current ones.

An issue strictly correlated with this topic is that, even if in an LRAAM it is possible to encode a cycle, what we get from the LRAAM is not an explicit reduced representation of the cycle, but several pointers to the components of the cycle forged in such a way that only implicitly the information on the cycle is represented in each of them. However, the ability to synthesize reduced descriptors for structures with cycles is what makes the difference between the LRAAM and the RAAM. The only system that we know of which is able to represent labeled graphs is the DUAL system proposed by Dyer [Dye91]. It is able to encode small labeled graphs representing relationships among entities. The idea is to have two networks, one responsible for the encoding of the relationships between one particular entity and the others, and one which devises a compressed representation of the weights of the first network. In this way, the compressed representation can be used in the first network as target for the training and, when information about the relationships of an entity with the others is needed, as pattern of activation for the hidden units of the second network to obtain the weights for the first network. However, this system cannot be considered at the same level of the LRAAM, since it devises a reduced representation of a set of functions relating the components of the graph instead of a reduced representation for the graph. Potentially also Holographic Reduced Representations [Pla91] are able to encode cyclic graphs.

The LRAAM model can also be viewed as an extension of the Hopfield networks philosophy. The basic idea is that, while Hopfield networks are able to exploit only minima of the associated energy function, the LRAAM is able to exploit the maxima as well. In fact, data stored in an LRAAM can be accessed both by pointer and by content. While access by pointer is a reliable procedure, access by content is not so reliable. However, recent developments in analog Hopfield networks with hidden units [AAM93] allow to test if an equilibrium state is asymptotically stable. Since the training set of an LRAAM defines, in first approximation, a set of equilibria states for the associated BAM, a prediction of the reliability of the access procedures can be made.

A relevant aspect of the use of the BAM associated to an LRAAM, is that the access procedures defined on it are able to exploit efficiently subsets of the weights. In fact, their use corresponds to generate several smaller networks from a large network, one for each kind of access procedure, each specialized on a particular feature of the stored data. Thus, by training a single network, we get several useful smaller networks.

6.1 Application of LRAAMs to neural dynamics control

Actually, the LRAAM model was developed as a way to synthesize a *neural code*, i.e., a set of weights which can be interpreted as a program for a particular recurrent neural network.

An example of neural code implementing a Neural Tree [SN90] has been given and different aspects of the neural code discussed in [Spe93, SS93a, SS93b]. Neural Trees (NTs) are decision trees where the decision at each node is taken by a perceptron[8]. Usually, the tree structure is stored and managed using classical symbolic data structures and programming. The joint use of π-connections and LRAAMs allows to implement a NT in a full neural architecture. The procedure to obtain this neural architecture is composed of three main steps:

[8] Or, in general, by a more complex neural network.

1. Represent the NT as a labeled tree, where the labels are the weights of the discriminators associated to the tree nodes;
2. Encode the NT in a LRAAM;
3. Load the weights of the LRAAM, i.e., the neural code, in a special network exploiting π-connections (the Executor Network) in order to "run" the NT encoded in the LRAAM.

In order to "run" the NT encoded in the LRAAM the Executor Network must be initialized with the pointer to the root of the NT. Then the flow of computation is driven by the input pattern, till a leaf of the NT is reached (terminal condition).

It is interesting to note that this method may optimize the number of parameters of the NT. In fact, if the dimension of the input patterns is far less than the number of nodes of the tree, linear redundancy in the weight space is present. This redundancy is automatically reduced by the LRAAM, since each pointer is a compressed version of the weights of a discriminator.

In this case only the storage properties of the LRAAM are used, but it would be interesting to study if the neural code generated in such a way subsumes other meaningful executions starting from pointers different than the pointer to the root of the NT.

Since the LRAAM is able to encode cyclic graphs it is not hard to imagine how to construct a neural code to generate a more complex dynamics. For example, a finite state machine (which in general contains cycles) can be encoded and executed using the same technique.

7 Conclusions

In this paper we have proposed the Labeling RAAM, an extension of the RAAM model by Pollack, which allows the encoding of labeled graphs with cycles. The LRAAM model allows also to solve in a more naturally way some technical problems of the RAAM. A theoretical analysis of the LRAAM under the hypotheses of perfect learning and linear output units showed cycles and confluent pointers impose constraints on the eigenvalues, eigenvectors and rank of the output weight matrices. These results bring an understanding on how to represent the training set in such a way to avoid useless constraints on the learning process. Examples of encoding of single structures has been presented, showing the proposed model to be able to synthesize not only structures with cycles, but also robust representations for the pointers, i.e., representations which do not degrade with recursive decoding along a cycle. However, more work on this issue must be done, in order to assess the condition under which such representations develop. A second relevant advantage of the LRAAM model is that it allows one to access data not only by pointer but also by content. In fact, it is possible to define an associated BAM which can be used under different operational modes in order to access data by different types of keys. The access by content is not wholly reliable. However, recent developments in analog Hopfield networks with hidden units can be used to foresee the reliability of the access procedures.

A study of the decoding part of an LRAAM with two hidden units has been presented in order to show the richness of the representations the model is able to develop. In particular, the partition of the pointer space is reminiscent of a fractal set, since information on deeper structures is codified in smaller and smaller partitions of the pointer space. Some representational effects related to the Hopfield-like nature of the decoding process have been noted, but more work is needed for a correct definition of their entity.

A sketch for the encoding of structures not present in the training set has been presented. The new encoding scheme is needed because of the use of multiple representations for the void pointer.

References

[AAM93] A. Atiya and Y. S. Abu-Mostafa. An analog feedback associative memory. *IEEE Transaction on Neural Networks*, 4:117–126, 1993.

[BMM92] D.S. Blank, L.A. Medeen, and J.B. Marshall. Exploring the symbolic/subsymbolic continuum: a case study of raam. In J. Dinsmore, editor, *The Symbolic and Connectionist Paradigms: Closing the Gap*, volume 1, pages 113–148. Lawrence Erlbaum, 1992.

[Cha90] D. J. Chalmers. Syntactic transformations on distributed representations. *Connection Science*, 2:53–62, 1990.

[Chr91] L. Chrisman. Learning recursive distributed representations for holistic computation. *Connection Science*, 3:345–366, 1991.

[Dye91] M. G. Dyer. *Symbolic NeuroEngineering for Natural Language Processing: A Multilevel Research Approach.*, volume 1 of *Advances in Connectionist and Neural Computation Theory*, pages 32–86. Ablex, 1991.

[Hin90] G. E. Hinton. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46:47–75, 1990.

[KL90] J. Kindermann and A. Linden. Inversion of neural networks by gradient descent. *Parallel Computing*, 14:277–286, 1990.

[Kos92] B. Kosko. *Neural Networks and Fuzzy Systems.* Prentice Hall, 1992.

[Pla91] T. Plate. Holographic reduced representations. Technical Report CRG-TR-91-1, Department of Computer Science, University of Toronto, 1991.

[Pol89] J. B. Pollack. *Implications of Recursive Distributed Representations*, pages 527–536. Advances in Neural Information Processing Systems I. San Mateo: Morgan Kaufmann, 1989.

[Pol90] J. B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2):77–106, 1990.

[Rei92] R. Reilly. A connectionist technique for on-line parsing. *Network*, 3:37–46, 1992.

[RT87] R. Rosenfeld and D. S. Touretzky. Four capacity models for coarse-coded symbol memories. Technical Report CMU-CS-87-182, Carnegie Mellon, 1987.

[Smo90] P. Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46:159–216, 1990.

[SN90] J. A. Sirat and J-P. Nadal. Neural trees: a new tool for classification. *Network*, 1:423–438, 1990.

[Spe93] A. Sperduti. *Optimization and Functional Reduced Descriptors in Neural Networks*. PhD thesis, Computer Science Department, University of Pisa, Italy, 1993. TD-22/93.

[SS93a] A. Sperduti and A. Starita. An example of neural code: Neural trees implemented by LRAAMs. In *International Conference on Neural Networks and Genetic Algorithms*, 1993. Innsbruck. To appear.

[SS93b] A. Sperduti and A. Starita. Modular neural codes implementing neural trees. In *6th Italian Workshop on Parallel Architectures and Neural Networks*, 1993. to appear.

[SW92] A. Stolcke and D. Wu. Tree matching with recursive distributed representations. Technical Report TR-92-025, International Computer Science Institute, 1992.

[Tou90] D. S. Touretzky. Boltzcons: Dynamic symbol structures in a connectionist network. *Artificial Intellicence*, 46:5–46, 1990.