Technical Report: TR-95-1, January 1, 1995

Dynamical Neural Networks Construction for Processing of Labeled Structures

Sperduti, A. Starita, A.

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

We show how Labeling RAAM (LRAAM) can be exploited to generate 'on the fly' neural networks for associative access of labeled structures. The topology of these networks, that we call Generalized Hopfield Networks (GHN), depends on the topology of the query used to retrieve information, and the weights on the networks' connections are the weights of the LRAAM encoding the structures. A method for incremental discovering of multiple solutions to a given query is presented. This method is based on terminal repellers, which are used to 'delete' known solutions from the set of admissible solutions to a query. Terminal repellers are also used to implement exceptions at query level, i.e., when a solution to a query must satisfy some negative constraints on the labels and/or substructures. Besides, the proposed model solves very naturally the connectionist variable binding problem at query level. Some results for a tree-like query are presented. Finally, we define a parallel mode of execution, exploiting terminal repellers, for the GHN, and we propose to use terminal attractors for implementing shared variables and graph queries.

ACM–CR Subject Classification:

Keyword and phrases: neural networks, processing of labeled structures

Università degli Studi di Pisa
Dipartimento di Informatica

Dynamical Neural Networks Construction for Processing of Labeled Structures

Alessandro Sperduti Antonina Starita
Dipartimento di Informatica
Corso Italia 40
56125 Pisa, Italy
E-mail:{perso,starita}@di.unipi.it

TR-1/95

Abstract

We show how Labeling RAAM (LRAAM) can be exploited to generate 'on the fly' neural networks for associative access of labeled structures. The topology of these networks, that we call Generalized Hopfield Networks (GHN), depends on the topology of the *query* used to retrieve information, and the weights on the networks' connections are the weights of the LRAAM encoding the structures. A method for incremental discovering of multiple solutions to a given query is presented. This method is based on *terminal repellers*, which are used to 'delete' known solutions from the set of admissible solutions to a query. Terminal repellers are also used to implement exceptions at query level, i.e., when a solution to a query must satisfy some negative constraints on the labels and/or substructures. Besides, the proposed model solves very naturally the connectionist variable binding problem at query level. Some results for a tree-like query are presented. Finally, we define a parallel mode of execution, exploiting terminal repellers, for the GHN, and we propose to use terminal attractors for implementing shared variables and graph queries.

Keywords: Neural Networks, Labeling RAAM, Hopfield network, associative memory, terminal repeller and attractor, structure processing, variable binding.

1 Introduction

The processing of structured information within connectionist systems is a relevant open research field in AI. In fact, even if connectionist systems have demonstrated their effectiveness in dealing with classification and approximation problems, they cannot easily deal with problems which require compositionality and systematicity of representation. These two features are fundamental for an effective and efficient solution of problems in complex domains. Moreover, these features can be exploited in full only by a system which is able to represent and to manage variables. Specifically, the problem of variable binding has been faced by several connectionist researchers, with results which still are not fully satisfactory.

In recent years, some interesting approaches to the representation of complex structures in connectionist systems have been proposed. For example, Hinton [Hin90] introduced the concept of distributed reduced descriptors in order to allow a neural network to represent compositional structures (see also the other papers of the same issue by J. B. Pollack, P. Smolensky and D. S. Touretzky). Concrete examples of distributed reduced descriptors are given by the Recursive Auto-Associative Memory (RAAM) by Pollack [Pol90] and the Holographic Reduced Representations by Plate [Pla91]. More recently, the Labeling RAAM model (LRAAM) [Spe93a, Spe94] has been proposed as an extension to the RAAM model, allowing the synthesis of distributed reduced descriptors for fixed-valence directed labeled graphs, and advances on the access by content capabilities of the LRAAM model has been discussed in [SS94].

In this paper, we propose an LRAAM-based framework for the encoding and processing of a set of labeled structures. The set of structures is encoded by an LRAAM which devises reduced descriptors for them (Section 2). The information encoded in the LRAAM can then be accessed in a structured way by building 'on the fly' recurrent networks (GHN) whose stable states correspond to the desired structures (or substructures), i.e., solutions to a given *query* (Section 3). These recurrent networks are obtained by properly *composing* subparts of the LRAAM, depending on the topology of the query which, in general, contains variables. We demonstrate how terminal repellers [Zak89] are particularly useful for retrieval of multiple solutions to a given query (Section 4.2). They are used to 'delete' solutions which have already been retrieved. Terminal repellers can also be used to implement exceptions at query level, thus providing an elegant solution to a particular instance of the variable binding problem (Section 5). Some experimental results are reported (Section 6). Finally, we propose to use terminal repellers for the definition of a parallel mode of execution for the GHN (*overloading*), i.e., a single network can simultaneously deal with multiple instances of the same query (Section 7), and terminal attractors for implementing shared variables and graph queries

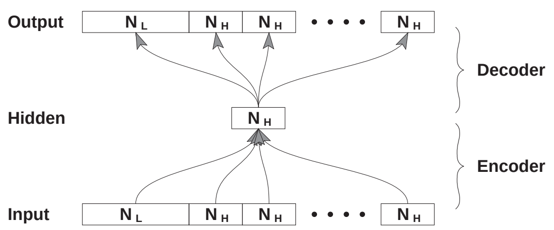

Figure 1: The network for a general Labeling RAAM.

(Section 8). Section 9 concludes the paper.

2 Labeling RAAM

The *Labeling RAAM (LRAAM)* [Spe93a, Spe93b, Spe94] is an extension of the RAAM model [Pol90] which allows one to encode labeled structures. The general structure of the network for an LRAAM is shown in Figure 1. The network is trained by backpropagation[1] to learn the identity function. The idea is to obtain a compressed representation (hidden layer activation) of a node of a labeled directed graph by allocating a part of the input (output) of the network to represent the label (N_L units) and the rest to represent one or more pointers. This representation is then used as pointer to the node. To allow the recursive use of these compressed representations, the part of the input (output) layer which represents a pointer must be of the same dimension as the hidden layer (N_H units). Thus, a general LRAAM is implemented by a $N_I - N_H - N_I$ feed-forward network, where $N_I = N_L + nN_H$, and n is the number of pointer fields.

Labeled directed graphs can be easily encoded using an LRAAM. Each node of the graph only needs to be represented as a record, with one field for the label and one field for each pointer to a connected node. The pointers only need to be logical pointers, since their actual values will be the patterns of hidden activation of the network. A graph is represented by a list of these records, and this list constitutes the initial training set for the LRAAM. During training the representations of the pointers are consistently updated according to the hidden activations. Consequently, the training set is dynamic. At the beginning of training the representations for the non-void pointers and void pointers in the training set are set at random. After each epoch, depending on the hidden activation obtained in the previous epoch for each pattern, the

[1]We verified faster learning using the *descending-epsilon* heuristic technique [YS90]: during the learning phase we maintain a list of the patterns having a decoding error higher than a specified threshold. The backpropagation is performed only on the patterns of the list: when all patterns are below the threshold, we lower it and resume the backpropagation.

representations for the non-void pointers in the training set are updated. The void representations are, on the other hand, copied from the output[2].

Once the training is complete, the patterns of activation representing pointers can be decoded to retrieve information. In order to decide whether a pointer is void or not, one bit of the label is allocated for each pointer field to represent the void condition. Notice that multiple labeled directed graphs can be encoded in the same LRAAM.

In the following, the terms *pointer*, *reduced descriptor*, and *reduced representation* will be used interchangeably.

3 Access by Content Using LRAAM

In Figure 2 we have summarized some of the concepts underpinning the access by content capabilities of LRAAM. At the top of the picture, an LRAAM network with one label and two reduced descriptor fields is shown. An example of encoding is given at the bottom left of the picture, where we have explicited the encoding process for a small tree. The information contained into a reduced descriptor can be accessed by using the decoder function implemented by the set of weights from the hidden to the output layer of the LRAAM. At the bottom center of the picture, the decoding process of the reduced descriptor is shown. Both the encoding and decoding processes are well defined only if the whole structure or a well formed reduced descriptor are given to them, respectively. If only partial information on the structure is available, then a recurrent network cyclically interleaving the encoding and decoding processes can be used to retrieve the missing information. An example of this network, that we call Generalized Hopfield Network (GHN), is given at the bottom right of the picture. Notice that, since multiple representations for the void pointers are allowed, the activity of fields representing *nil* pointers is obtained by decoding the current reduced descriptor. A random pattern is used to initialize the field(s) for which no information is available and the network left free to relax on a stable state. A stable state represents a *valid solution* only if its reduced descriptor is well formed, i.e., if the decoding process applied to the reduced descriptor is consistent with the known information. In this example, the partially defined structure constitutes a *query* and each field for which no information is available can be considered as a *variable* of the query. We omit other implementational details. It is sufficient to point out that variables can appear not only in label fields, but also in reduced descriptor fields, i.e., the query may have a reduced descriptor field which can be instantiated either to a non-void reduced descriptor or to a void descriptor.

A GHN is well defined only with tree-like queries, since the LRAAM network allows the encoding of a single component at time. This means that only

[2] This allows the network to develop multiple representations for the void pointer.

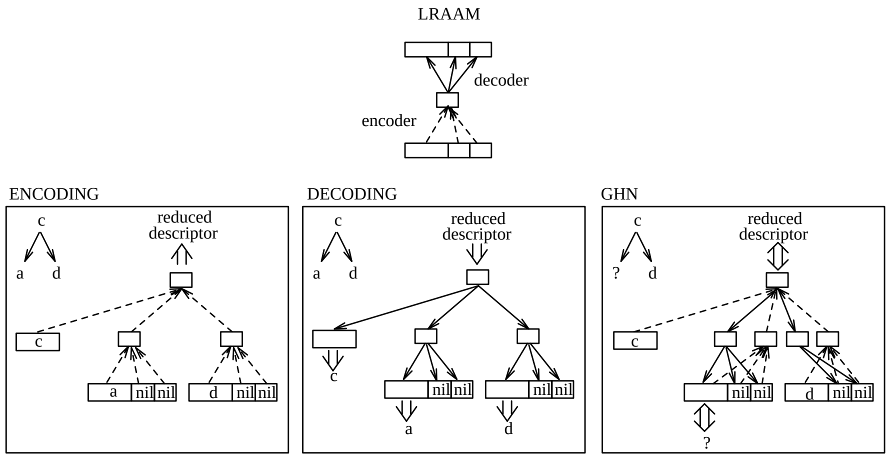

Figure 2: **Bottom left and center:** encoding and decoding of a small labeled tree using an LRAAM (top); **bottom right:** the GHN network for a partially defined tree.

queries which have components with a single entering pointer can be implemented by a GHN. However, in Section 8 we will propose a technique for the implementation of graph queries.

4 Multiple Solutions

In general a query can have multiple solutions. Multiple solutions correspond to multiple memories in the associated GHN. Below we discuss how to draw different multiple solutions from the GHN. Two different approaches are suggested and analyzed.

4.1 Uniform Random Search

The simplest way to explore the space of solutions of a GHN is to repeat the relaxation process, provided that the top layer of the GHN is initialized with a reduced descriptor drawn from a uniform random distribution. This approach works well if all the solutions have the basins of attraction roughly of the same size. In fact, in this case, the solutions are uniformly distributed as well. However, the learning algorithm used to train the LRAAM does not guarantee that this condition holds, and usually it does not hold. Thus, if a solution has a very small basin of attraction with respect to the basins of attraction of other solutions, then the probability to find it will be small as well.

The problem with this approach is that, when searching for a new solution, no information on the already found solutions is used. A successful approach should be such that as soon as a solution is discovered, its probability to be found again becomes null at advantage of unknown solutions. A good approximation to this ideal scheme, based on the concept of terminal repeller, is discussed in the next section.

4.2 A Search Procedure Exploiting Terminal Repellers

In the following, we briefly introduce *terminal attractors* and *terminal repellers*. Terminal repellers are fundamental for the development of a new searching procedure able to "delete"already found solutions from the searching space.

4.2.1 Terminal Attractors and Repellers

Terminal attractors and terminal repellers were first introduced by Zak [Zak89]. The peculiarity of a terminal attractor is that it is "infinitely"stable, while a terminal repeller is "infinitely"unstable. In standard recurrent neural networks used for content addressable memory, such as:

$$\dot{x} = f_i(x_1, x_2, \ldots, x_n),\ 1, 2, \ldots, n, \tag{1}$$

terminal attractors and repellers are not present, since any network satisfies the Lipschitz condition

$$\left|\frac{\partial f_i}{\partial x_j}\right| < \infty. \tag{2}$$

The Lipschitz condition guarantees the existence of a unique solution for each of initial conditions. Because of that, the theoretical time of approaching the attractor and escaping the repeller is always infinite. Terminal attractors and repellers, instead, violate the Lipschitz condition, being consequently reached and escaped in finite time.

A simple example of terminal repeller is given by the following equation:

$$\dot{y} = y^{1/3}, \tag{3}$$

where the equilibrium point $y = 0$ violates the Lipschitz condition:

$$\frac{d\dot{y}}{dy} = \frac{1}{3}y^{-2/3} \to \infty \text{ at } y \to 0. \tag{4}$$

Moreover, since

$$Re(\lambda) \to \infty > 0, \tag{5}$$

where $Re(\lambda)$ is the real part of the eigenvalue defined for the linearized system at $y = 0$, the point is a repeller of "infinite"unstability, i.e., if the initial condition is infinitely close to this repeller, the transient solution will escape the repeller during a finite time period.

Similarly, an example of terminal attractor is given by the equation:

$$\dot{y} = -y^{1/3}. \tag{6}$$

It is easy to verify that in this case the equilibrium point $y = 0$ is an attractor of "infinite"stability. For more details we remand to the paper by Zak [Zak89].

4.2.2 The Searching Procedure

Terminal repellers can be used to modify the dynamical features of our GHNs. Specifically, since a terminal repeller is infinitely unstable, it can turn an asymptotically stable fixed point of a GHN into an unstable one. This effect can be obtained by adding to the equations of the network an appropriate term violating the Lipschitz condition. In this way, we are able to "delete"solutions we already know of. For the sake of presentation, in the following we use continuous differential equations. Their actual implementation, however, was obtained by using their associated difference equations.

Let Q be a query to our system and GHN_Q to be the network implementing the query. Given the reduced descriptor $p^{(s)}$ of a solution s to the query Q, i.e., an asymptotically stable fixed point of GHN_Q, the equations of the top layer of GHN_Q can be modified from:

$$\dot{x}_i^{(top)} + x_i^{(top)} = \sum_{j=1}^{N_O} E_{ij} f(x_j^{IN(top)}), \tag{7}$$

to:

$$\dot{x}_i^{(top)} + x_i^{(top)} = \sum_{j=1}^{N_O} E_{ij} f(x_j^{IN(top)}) + \alpha (x_i^{(top)} - p_i^{(s)})^{1/3} e^{-\gamma(\sum_{i=1}^{N_H}(x_i^{(top)} - p_i^{(s)})^2)}, \tag{8}$$

where i varies from 1 to N_H (the number of hidden units in the LRAAM), N_O is the number of units to which the top layer of the GHN feeds, $f()$ is a sigmoidal function, E is the weights matrix implementing the encoder in the LRAAM, $x^{(top)}$ is the state vector of the units implementing the reduced descriptor at the top layer of the query, and $IN(top)$ refers to the set of units feeding the top layer. The term $(x_i - p_i^{(s)})^{1/3}$ violates the Lipschitz condition, introducing a terminal repeller at location $p^{(s)}$; the exponential multiplier depends on the distance of x from $p^{(s)}$ and it is introduced in order to localize the effect of the terminal repeller. The basin of repulsion of the repeller is controlled by the coefficients α and γ which must be tuned so that other solutions are not affected by it.

The resulting system has the same solutions as the original one[3], except for the solution s. This system can be used to search for another solution s' and if such solution is found, a new terminal repeller can be added in order to "delete" s' from the space of solutions. Given a set of solutions $\{s^0, \ldots, s^M\}$ the equations for the top layer of the GHN_Q used to retrieve a new solution becomes:

$$\dot{x}_i^{(top)} + x_i^{(top)} = \sum_{j=1}^{N_O} E_{ij} f(x_j^{IN(top)}) + \alpha \sum_{k=0}^{M} (x_i^{(top)} - p_i^{(s^k)})^{1/3} e^{-\gamma(\sum_{i=1}^{N_H}(x_i^{(top)} - p_i^{(s^k)})^2)}. \tag{9}$$

[3] Here we are referring to the solutions to the query, not to the solutions of the differential equations.

In order to avoid that a terminal repeller might destroy a fixed point corresponding to another solution, the γ parameter must be tuned carefully. Specifically, the effect of the gaussian controlled by the γ parameter, must vanish when considering other fixed points. This can be obtained by setting

$$\gamma = -\frac{log(\epsilon)}{d_{min}^2}, \tag{10}$$

where ϵ is a small positive constant (e.g., $0 < \epsilon < 10^{-5}$) and d_{min} is the minimum distance among the reduced descriptors of structures encoded by the LRAAM:

$$d_{min} = \min_{ij} \ \{d_{ij} | d_{ij} = \|p^{(i)} - p^{(j)}\|, \text{ and } p^{(i)},\ p^{(j)} \text{ are in the training set}\}. \tag{11}$$

One problem with this kind of scheme is that the introduction of several terminal repellers can introduce cycles or chaotic dynamics in the network. In fact, it can happen that a trajectory is regularly or chaotically repelled by the same set of repellers, without having the opportunity to converge to a new solution. An event of this kind may be overcome by changing the initial conditions for the trajectory. However, since it is intractable to recognize whether we are in presence of such events, we are not able to decide when to stop and reinitialize the dynamics of the system. For this reason we introduce a decay in the coefficient α which guarantees that after a given amount of time the effect of the repellers is removed:

$$\dot{\alpha} = -\frac{1}{\tau}\alpha, \tag{12}$$

where $\tau \gg 1$ is a constant. In this way, the system will eventually relax on an 'old' solution or, if we are lucky, on a new one. If the system relax on an old solution s^q, another repeller for that solution can be introduced. In this way the coefficient α for that solution will be doubled as well as the repulsion strength of the associated repeller. Consequently, the probability that the same solution s^q is found at the next trial decreases.

5 Exceptions

Terminal repellers can also be used to extend the kind of queries which can be formulated in our framework. In fact, they allow to implement queries with exceptions. An exception $(val, node_{id}, field_{id})$ is a constraint on the values that a solution to the query can take in a given field, i.e., the value of the field $field_{id}$ in the node $node_{id}$ must be different from val. Multiple exceptions can be given for the same field. A trivial way to implement a query with exceptions is to remove the exceptions, to find all the solutions, and then to remove the solutions which do not satisfy the exceptions. This procedure is however inefficient, since more solutions than necessary must be found. Below we discuss how terminal repellers can be used to search only for solutions which satisfy the exceptions.

5.1 Label and Reduced Descriptor Exceptions

An effective implementation of a label exception (val, n_i, l) can be obtained by introducing a local *detector* for val into the label field l of node n_i. The activation of the detector triggers the creation of the following terminal repeller at the top layer of the GHN:

$$\alpha(x_i^{(top)} - x*_i^{(top)})^{1/3} e^{-\gamma(\sum_{i=1}^{N_H}(x_i^{(top)} - x*_i^{(top)})^2)}, \tag{13}$$

where $x*^{(top)}$ is the activation vector of the top layer whose decoding led to the activation of the detector[4].

Notice that, the insertion of terminal repellers at the top layer, limiting the search space to solutions that satisfy the exceptions contained in the query, is particularly powerful when multiple solutions to the query are present. In fact, the terminal repellers introduced during the search for the first solution can be retained so that the search for successive solutions will not require to re-compute them. The overall effect is that, after a given number of iterations, which is query dependent, the system will perform the search only in zones of the representational space which satisfy the exceptions.

Exceptions can also involve reduced descriptors. Reduced descriptors are useful since they represent in a very compact way structured information. This property can be exploited when specifying a query with an exception which is not limited to a label field, but involves a whole subtree. In this case a reduced descriptor exception is introduced. Suppose that p is the reduced descriptor of the subtree which is not desired to appear in a given position into solutions of the query. The exception will have the form (p, n_s, p_m), where p_m is the reduced descriptor field of node n_s for which we want to forbid the value p. The implementation of the exception proceeds as in the case of label exceptions with the introduction of a local detector for p into the field p_m of n_s, and with the generation of an opportune repeller as soon as the detector is activated.

Note that multiple reduced descriptor exceptions can be implemented by adding the single exceptions to the equations of the dynamics. Moreover, label and reduced descriptor exceptions are not mutually exclusive, so that both can appear in a query.

---

[4]The detector can be implemented in different ways. For example, it can became active if the distance between the vector representing the current status of the label field of node n_i and val is below a predefined threshold, or (when the labels' components are in the set $\{-1, 1\}$) if the sign of the first vector agrees componentwise with the sign of the second vector.

6 Examples of Queries

In this section we show how the expected number of solutions to a query for a given sequence of 'runnings' of the corresponding GHN is modified by the introduction of terminal repellers.

A $34-13-34$ LRAAM was trained on the training set shown in Figure 3. The label field was composed by $6+2$ units: 6 units were used to represent the different labels using a localist representation, and the other 2 units were used to represent the void condition for the reduced descriptor fields. The training took 16600 epochs to reach a maximum error for a single output unit below 0.035. The obtained network was able to encode and to decode the training set.

In order to allow a large number of different solutions, we tested the query shown at the top of Figure 4. All the label fields contain a variable and the only known information concerns the structure of the query. The structure, however, is left not defined below the sons of the root of the query. This corresponds to have variables also for the reduced descriptor fields of the sons. In the figure, we show the set of admissible solutions to the query, plus a crossed box that represents solutions of the GHN that did not pass the validation process, i.e., ill-formed solutions.

The basin of attraction of each solution was evaluated by 8192 runnings of the GHN, where for each running the top layer of the GHN was set to a different vertex of the hypercube in 13 dimensions (the hidden layer dimension). The results are shown, in percentage, below each solution (**v:**). Then, using a pseudo-random number generator implemented in `C`, we executed another 8192 runnings of the GHN and used the results to compute the expected number of a given solution into a sequence of 20 runnings (**r-20:**). Finally, we executed 50 sequences of 20 runnings each, using the terminal repeller technique. The terminal repellers were retained only inside a single sequence. All the terminal repellers were removed before to start a new sequence. The expected number of a given solution obtained with this procedure is shown in figure (**t-20:**). The results shown the effectiveness of the technique, since the ideal distribution is given by one single occurrence for each well-formed solution and 8 occurrences of ill-formed solutions. During the last set of runnings the γ parameter was set

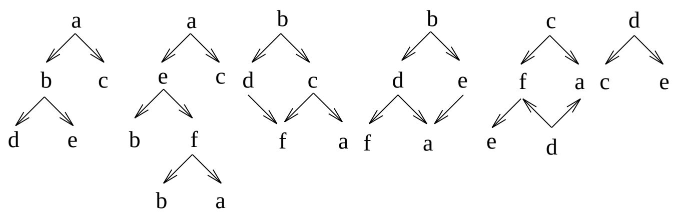

Figure 3: The training set for the LRAAM.

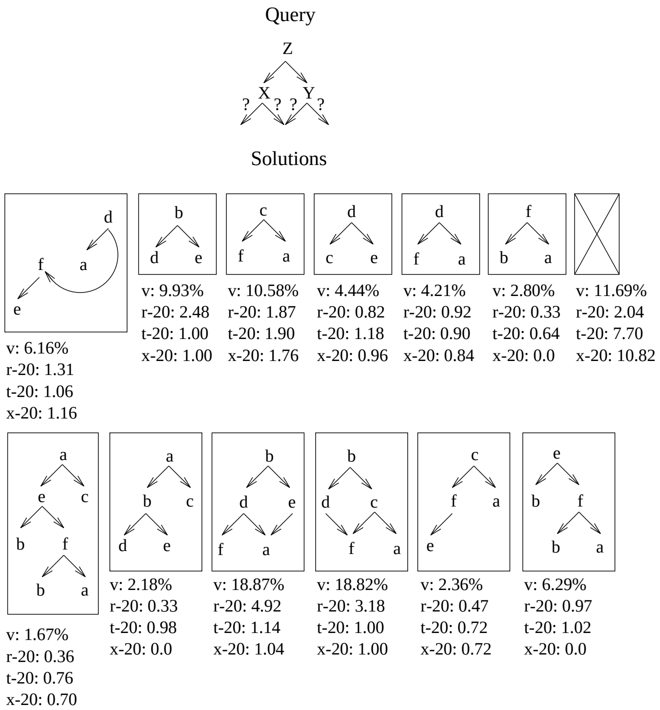

Figure 4: Results obtained for the query at the top of the picture.

to 3.4 using eq.(10), and α initializated to 200 and decremented of one unity at each iteration of the GHN. The results obtained by introducing also the exception **b** for the X variable are reported as well (**x-20:**). The exception technique works very well, since it has also minor effects on the expected number of solutions which are not removed by the exception. In our experiments, however, we observed the introduction of up to 150 repellers, many of whom were very similar. We are working on a variant of the method which introduces a repeller only if needed.

7 Overloading and Parallel Processing

Neural networks implementing associative memory are often criticized for the fact that a stable state of the network represents a single solution, and consequently only one solution at time can be found. In our case, we have also the problem that the complexity of one iteration of the GHN depends on the depth of the query. Specifically, if the query consists of k levels, then each iteration of the GHN takes $2k$ steps, where each step consists of a multiplication of a vector state with a matrix of weights followed by a nonlinear transformation of the result. Therefore, deeper queries will correspond to longer computations and the retrieval of multiple solutions will become even more expensive. However, if we observe that the model is synchronous and no unit employs short term memory, then it is easy to recognize that the network can be overloaded,

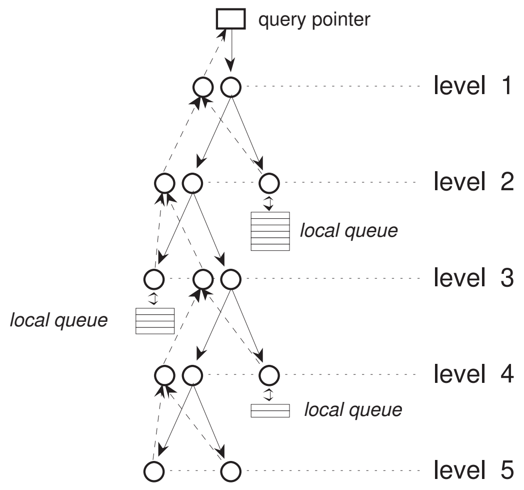

Figure 5: Example of a modified GHN network for *overloading* management. Memory queues are added at terminal nodes in intermediate levels.

i.e., more computations can be *piped* in it.

In particular, $2k$ different computations for the same query can be processed by the same network, provided that the computations are shifted in time of one step: the $2k$-th computation will begin to be processed simultaneously with the conclusion of the first iteration of the first computation. In order for the overloading to work properly, each layer of neurons implementing a leaf of the query at level r needs a memory queue of length $2(k-r)$ where to store temporary results (see Figure 5). These memory queues are necessary to maintain the correct synchronization between single steps of the same computation.

The overloading of different computations for the same query can be exploited to explore the solution space of the query. It allows also a fully exploitation of the terminal repeller technique since, as soon as a computation converges to a valid solution, a terminal repeller affecting also the other computations can be introduced into the top layer of the network. In order to avoid cycles and chaotic dynamics, different coefficients α can be introduced for each solution, so that the earlier introduced repellers will be the first ones to decay.

A refinement of this scheme considers the presence of a repulsive force not only for each stable computation, but also among the current activation states of different computations. These repulsion forces can be implemented into the top layer of the network according to the following equations, where we have omitted the reference to the top layer for the sake of notation:

$$\dot{x}[h]_i + x[h]_i = \sum_{j=1}^{N_O} E_{ij} f(x[h]_j^{IN}) + \sum_{r=1, r \neq h}^{2k} \alpha_r (x[h]_i - x[r]_i)^{1/3} e^{-\gamma(\sum_{i=1}^{N_H}(x[h]_i - x[r]_i)^2)}, \tag{14}$$

where $x[h]$ represents the activation state of the h-th computation at the top layer, and $x[r]$ is picked up for each value of r from a queue of length $2k$ associated to the top layer. As soon as a computation becames stable, i.e., $\dot{x}[h] = 0$, the associated coefficient α_h starts to decay to zero.

Overloading can also be extended to simultaneously manage multiple queries with the same topology. The idea is to provide each input layer of the network with a memory queue as well, where all the different keys used in the access procedures are stored and consistently read during processing.

8 Shared Variables and Graph Queries

In this section, we discuss how terminal attractors may be used for implementing shared variables and graph queries. The basic idea is very simple: multiple sets of units are maintained for each occurrence of the same variable and/or pointer, but the activations of sets of units representing the same shared entity[5] are forced to become equal by the introduction of terminal attractors. For example, suppose the pointers p_m, at node n_s, and p_k, at node n_r, refer to the same pointer; then the following equations are used:

$$\dot{x}_i^{(p_m,n_s)} + x_i^{(p_m,n_s)} = \sum_{j=1}^{N_H} D_{ij}^{(p_m)} f(x_j^{IN(p_m,n_s)}) - \alpha(x_i^{(p_m,n_s)} - x_i^{(p_k,n_r)})^{1/3}, \tag{15}$$

$$\dot{x}_i^{(p_k,n_r)} + x_i^{(p_k,n_r)} = \sum_{j=1}^{N_H} D_{ij}^{(p_k)} f(x_j^{IN(p_k,n_r)}) - \alpha(x_i^{(p_k,n_r)} - x_i^{(p_m,n_s)})^{1/3}, \tag{16}$$

where $D^{(p_m)}$ is the $N_H \times N_H$ decoding matrix for p_m and $D^{(p_k)}$ the $N_H \times N_H$ decoding matrix for p_k[6]. Notice that, this time the terminal attractors are used to modify the decoding dynamics of the LRAAM and not the encoding dynamics. This is required by the constraint that only the single entity must be affected, i.e., the attractors must work independently of the local context involving the entity. In this way, $x^{(p_m,n_s)}$ and $x^{(p_k,n_r)}$ attract each other in a finite time, and consequently will relax on the same status satisfying as well the other constraints imposed by the query. Similar equations can be introduced in the case of shared variables.

[5]In the following, we will use the term *entity* when referring either to a variable or a pointer.

[6]Both $D^{(p_m)}$ and $D^{(p_k)}$ correspond to submatrix of D, i.e., the LRAAM's decoding matrix.

If more than two entities need to be equal, terminal attractors can be introduced in such a way to form a closed chain. Using this approach it is possible to reduce the number of terms that must be added to the equations of the dynamics. Suppose entities e_j with $j = 1, 2, \ldots, K$ need to be equal, then we use the set of equations:

$$\dot{x}_i^{(e_1)} = G_{(i,e_1 \rightarrow e_2)}(x_i^{(e_1)}, x_i^{(e_2)}), \tag{17}$$

$$\dot{x}_i^{(e_2)} = G_{(i,e_2 \rightarrow e_3)}(x_i^{(e_2)}, x_i^{(e_3)}), \tag{18}$$

$$\vdots$$

$$\dot{x}_i^{(e_K)} = G_{(i,e_K \rightarrow e_1)}(x_i^{(e_K)}, x_i^{(e_1)}), \tag{19}$$

where $G_{(i,e_r \rightarrow e_q)}$ is a short notation for the function that introduces a terminal attractor between entity e_r and entity e_q in the standard GHN equations: e.g., for Equation (15) we have

$$G_{(i,p_m \rightarrow p_k)} = -x_i^{(p_m,n_s)} + \sum_{j=1}^{N_H} D_{ij}^{(p_m)} f(x_j^{IN(p_m,n_s)}) - \alpha(x_i^{(p_m,n_s)} - x_i^{(p_k,n_r)})^{1/3}. \tag{20}$$

Of course, in order to compute $G_{(i,e_r \rightarrow e_q)}$, the pool of units implementing e_r needs to know the status of the pool of units implementing e_q, i.e., a proper set of connections between the two pools must be introduced. These connections can be predefined and enabled only and only if a terminal attractor is introduced.

9 Conclusions

In this paper, we have proposed an LRAAM-based framework for the processing of labeled structures. We showed that an LRAAM, encoding a set of structures, can be used as basic building block for the dynamical construction of Generalized Hopfield Networks. A Generalized Hopfield Network allows the associative access of the structures through a query and it is constructed 'on the fly', depending on the topology of the query and using the weights of the LRAAM. We have demonstrated the feasibility of using terminal repellers in order to obtain a uniform recall probability for all the solutions to a query. Preliminary results seems to suggest that this technique is able to speed up the retrieval of different solutions if them are not uniformely distributed. Terminal repellers proved also effective for the implementation of exceptions.

Moreover, we defined a parallel mode of execution, exploiting terminal repellers, for the GHN, and we proposed to use terminal attractors for implementing shared variables and graph queries. Both of this issues will be the subject of future researches. It must be pointed out that the use of terminal repellers is justified both by their dynamical behavior and their modularity. On one hand, they are able to turn *any stable state* of the GHN into an unstable one, and they can be escaped in *finite time*. On the other hand, they can be introduced in the GHN incrementally, without destroying the global structure of the GHN's dynamics.

Finally, we would like to notice that our model solves very naturally the connectionist variable binding problem at query level, since a variable in the query is dynamically bound to an entity (label or reduce descriptor) during retrieval. The nature of the binding is not static. In fact, in the case of exceptions, the binding of a variable can be modified dynamically because of the introduction of a terminal repeller at the top layer of the GHN.

References

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

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

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

[Spe93a] A. Sperduti. Labeling RAAM. Technical Report 93-029, International Computer Science Institute, 1993. (a revised version will appear on Connection Science).

[Spe93b] A. Sperduti. On some stability properties of the LRAAM model. Technical Report 93-031, International Computer Science Institute, 1993. A revised version will appear on IEEE Transactions on Neural Networks.

[Spe94] A. Sperduti. Encoding of Labeled Graphs by Labeling RAAM. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 1125–1132. San Mateo, CA: Morgan Kaufmann, 1994.

[SS94] A. Sperduti and A. Starita. On the Access by Content Capabilities of the LRAAM. In *IEEE World Congress on Computational Intelligence, Conference on Neural Networks*, pages 1143–1148, 1994.

[YS90] Y. Yu and R. Simmons. Descending epsilon in backpropagation: a technique for better generalization. In *International Joint Conference on Neural Networks*, pages 167–172, 1990.

[Zak89] M. Zak. Terminal attractors in neural networks. *Neural Networks*, 2:259–274, 1989.