Chapter 1

A Tutorial on Neurocomputing of Structures

A Sperduti
Department of Computer Science
University of Pisa
perso@di.unipi.it

Abstract

In this chapter we present a tutorial on neural networks for the processing of structured information. Basic concepts on structured domains are first introduced. Then earlier work on representing data structures within traditional neural networks is briefly reviewed. This review introduces some basic concepts which are exploited to realize graph transductions. Specifically, we discuss neural realizations of a specific class of graph transductions for classification of labeled acyclic graphs. We show how the standard definition of neuron and neural learning algorithms can be extended to deal with structured domains. A specific section is then devoted to the case of cyclic graphs. In the second part of the tutorial, computational and complexity results concerning this class of neural networks are reviewed. Almost all these results exploit Tree Automata Theory as medium

to assess the computational capabilities and functional complexity of this kind of neural networks.

1.1 Introduction

Structured representations are ubiquitous in different fields such as knowledge representation, language modeling and pattern recognition. Examples of application domains where structures are extensively used are medical and technical diagnoses, molecular biology, chemistry, automated reasoning, and so on.

While algorithms that manipulate symbolic information are capable of dealing with highly structured data, adaptive neural networks are mostly regarded as learning models for domains in which instances are organized into *static* data structures, like records or fixed-size arrays. Recurrent neural networks, that generalize feedforward networks to sequences (a particular case of dynamically structured data) are perhaps the best known exception.

The interest in developing connectionist architectures capable of dealing with these rich representations (as opposed to "flat" or vector-based representations) can be traced back to the end of the 80's [Tou90, Pol90, Hin90, Pla95, Smo90]. However, only during the last few years a general framework for the development of neural networks for the representation and processing of structures have been developed [SS97, FGS98]. In particular, these neural networks are a generalization of recurrent networks for processing sequences (i.e., linear chains from a graphical point of view) to the case of (mainly) directed acyclic graphs.

There are several reasons for being interested in processing of structures by neural networks. First of all, neural networks are universal approximators. Moreover, they are good classifiers and they are robust to the presence of noise. These capabilities are important when considering real-world applications in structured domains. In order to exemplify, let consider the classification of labeled graphs which represent chemical compounds. The standard approach consists in encoding each graph X as a fixed-size vector which is then given as input to a feed-forward neural network for classification. This approach is motivated by the fact that neural networks only have a fixed number of input units while graphs are variable in size. The encoding process is usually defined a priori and does not depend on the classification task, e.g., in molecular biology and chemistry, the encoding process is performed through the definition of *topological indexes* [HK91], which are designed by a very expensive trial and error approach.

The a priori definition of the encoding process faces the *specificity-*

generality dilemma, i.e., if the encoding is too specific, new features must be devised (by suitable experts) for each new computational problem; on the other hand, if the encoding is too general, the representing features are weakly relevant with respect to the specific target and so they turn out to be difficult to classify. To overcome the dilemma, the encoding process should be adapted automatically to the specific classification task at hand. This can be done by implementing the encoding process through an additional neural network which is trained, alongside the neural network performing the classification, to learn the best way to encode the graphs for the given classification task.

Of course, neither standard neurons nor recurrent neurons are powerful enough to be used in such a network. This is because the former were devised for processing unstructured patterns, while the latter can only naturally process sequences. However, by extending the concept of recurrent neuron it is possible to formalize several supervised models for classification of structures which stem very naturally from well known models, such as Back-Propagation Through Time networks, Real-Time Recurrent networks, Simple Recurrent networks, Recurrent Cascade Correlation networks, and Neural Trees.

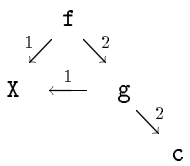
When developing a new computational model, it is important to understand its computational power and complexity. This evaluation is typically done by resorting to known results in related models. For example, it is well known that recurrent neural networks can simulate any finite state automata [ADO91, OG96] as well as any multi-stack Turing machine in real time [SS95]. When constraining the network architecture, however, this computational power may no longer hold. For example, Elman-style Recurrent Networks can simulate any finite state automata [GGCC94, Kre95], while Recurrent Cascade-Correlation cannot [GCS$^+$95, Kre96]. Here we discuss the computational capabilities of Elman-style Recurrent Networks, Recurrent Cascade Correlation networks, and Neural Trees with respect to the classification of structures. We will relate them to Frontier-to-Root Tree Automata (FRA) [Tha73, GT78] and show that Elman-style Recurrent Networks can simulate any FRA, while neither Cascade-Correlation networks, nor Neural Trees can. Moreover, we briefly report a known result which states that, if the set of possible labels (alphabet) is finite, then any mapping from trees to the set of reals can be implemented by a neural network for structures having a sufficient number of parameters.

Computational complexity of the model is important as well. Here we give both upper and lower bounds on the number of neurons (or connections) needed to implement an arbitrary frontier-to-root Tree Automata into the most powerful model. Some results on the complexity of learning are sketched as well.

1.2 Basic Concepts

Here we consider structured domains which are sets of annotated directed ordered graphs (DOGs). For a DOG we mean a directed graph $\boldsymbol{Y}$ with vertex set $\mathrm{vert}(\boldsymbol{Y})$ and edge set $\mathrm{egd}(\boldsymbol{Y})$, where for each vertex $v \in \mathrm{vert}(\boldsymbol{Y})$ a total order on the edges leaving from v is defined. Moreover, vertices are annotated by labels which are tuples of variables. The void DOG will be denoted by the special symbol ξ.

For example, in the case of graphs representing logical terms with variables (denoted by capital letters), the order on outgoing edges is immediately induced by the order of the arguments to a function; e.g., the logical term `f(X,g(X,c))` can be represented as



Note that, since the `X` variable is shared, a single node is used to represent it.

In problems of structure classification, we shall require the DOG either to be empty or to possess a supersource, i.e. a vertex $s \in \mathrm{vert}(\boldsymbol{Y})$ such that every vertex in $\mathrm{vert}(\boldsymbol{Y})$ can be reached by a directed path starting from s. Note that if a DOG does not possess a supersource, it is still possible to define a convention for adding an extra vertex s (with a minimal number of outgoing edges), such that s is a supersource for the expanded DOG [SS97]. When considering a DOAG, i.e., an acyclic DOG, the function $\mathit{source}(\boldsymbol{Y})$ returns the (unique) supersource of $\boldsymbol{Y}$. We also assume that a convention is defined in such a way that, given a cyclic DOG, the function $\mathit{source}(\boldsymbol{Y})$ returns a predetermined supersource within the set of supersources[1].

Given a DOAG $\boldsymbol{Y}$ and $v \in \mathrm{vert}(\boldsymbol{Y})$, we denote by $\mathrm{pa}[v]$ the set of parents of v, by $\mathrm{ch}[v]$ the set of children of v, by $\mathrm{de}[v]$ the set of descendants of v, and by $\mathrm{an}[v]$ the set of ancestors of v. We will use the notation $\mathrm{ch}_j[v]$ to refer to the jth children of v. The *indegree* of v is the cardinality of the set $\mathrm{pa}[v]$ the *outdegree* of v is the cardinality of the set $\mathrm{ch}[v]$. In the following, we shall denote by $\#^{(i,o)}$ the class of DOAGs with maximum indegree i and maximum outdegree o. In our logical terms example, the maximum outdegree corresponds to the maximum arity of the functions being considered, e.g., the maximum outdegree of `f(X,g(X,c))` is 2. A generic class

[1] A cyclic graph may have several supersources.

of DOAGs with bounded (but unspecified) indegree and outdegree, will simply be denoted by #.

Subscript notation will be used when referencing the labels attached to vertices in a data structure. Hence $\boldsymbol{Y}_v$ denotes the set of variables labeling vertex $v \in \text{vert}(\boldsymbol{Y})$.

Given a data structure $\boldsymbol{Y}$, the DOG obtained by ignoring all node labels will be referred to as the *skeleton* of $\boldsymbol{Y}$, denoted skel($\boldsymbol{Y}$). Clearly, any two data structures can be distinguished because they have different skeletons, or, if they have the same skeleton, because they have different node labels. The class of all data structures defined over the label universe domain $\mathcal{Y}$ and skeleton in $\#^{(i,o)}$ will be denoted as $\mathcal{Y}^{\#^{(i,o)}}$.

We define a *structured domain* $\mathcal{D}$ (over the set of labels Σ) as any (possibly infinite) set of graphs (over Σ). The *valence of a domain* $\mathcal{D}$ is defined as the maximum among the out-degrees of the graphs belonging to $\mathcal{D}$.

Since we will deal with learning, we need to define the target function we want to learn. In approximation tasks, a *target function* $d()$ over $\mathcal{D}^{\#}$ is defined as any function $d : \mathcal{D}^{\#} \rightarrow I\!R^k$, where k is the output dimension, while in (binary) classification tasks we have $d : \mathcal{D}^{\#} \rightarrow \{0,1\}^k$ (or $d : \mathcal{D}^{\#} \rightarrow \{-1,1\}^k$.) A training set T on a domain $\mathcal{D}^{\#}$ is defined as a set of couples $(\boldsymbol{X}, d(\boldsymbol{X}))$, where $\boldsymbol{X} \in \mathcal{U}^{\#} \subseteq \mathcal{D}^{\#}$ and $d()$ is a target function defined on $\mathcal{D}^{\#}$.

Finally, for the sake of completeness, here we recall the basic definitions of standard and recurrent neurons. The output $o^{(s)}$ of a standard neuron is given by

$$o^{(s)} = f(\sum_i w_i I_i), \tag{1.1}$$

where *f()* is some non-linear squashing function applied to the weighted sum of inputs I[2]. A recurrent neuron with a single self-recurrent connection, on the other hand, computes its output $o^{(r)}(t)$ as follows

$$o^{(r)}(t) = f(\sum_i w_i I_i(t) + w_s o^{(r)}(t-1)), \tag{1.2}$$

where *f()* is applied to the weighted sum of inputs I plus the self-weight, w_s, times the previous output. The above formula can be extended both considering several interconnected recurrent neurons and delayed versions of the outputs [Hay94].

[2] The threshold of the neuron is included in the weight vector by expanding the input vector with a component always to 1.

1.3 Representing Structures in Neural Networks

Structures may be represented in a neural network by using either a *localist representation* or a *distributed representation*. In the localist representation each neuron represents a component of the structure, while each connection represents a relation between components. This way of representing structures is trivial and not efficient, since different structures needs to be represented by different networks. For this reason, several researchers have developed neural systems using different types of distributed representation, such as in DCPS [TH88], BoltzCONS [Tou90], Tensor Products [Smo90], RAAM [Pol90] and LRAAM [Spe94a, Spe94b], Convolution-based models (CHARM [Met91], TODAM2 [Mur93]), Holographic Reduced Representations [Pla95]. In the following we discuss the RAAM family of models, since they can be considered the precursors of the models presented in this chapter.

1.3.1 The RAAM Family

The task of a network in the RAAM family [SS96] consists in devising a reduced descriptor for each structure (and substructure) in a given set of structures, i.e., the training set. The concept of reduced descriptor was first introduced by Hinton [Hin90]. The basic idea is to represent complex conceptual structures in distributed representations. These representation, however, should satisfy four desiderata

1. **Representational adequacy:** it should be possible to reconstruct the full representation of the structure from the reduced descriptor;

2. **Reduction:** the reduced descriptor should require the allocation of fewer units then the full explicit representation of the structure;

3. **Systematicy:** the reduced descriptor should be related in a systematic way to the full representation;

4. **Informativeness:** the reduced descriptor should tell us something about the concept that it refers to, without it being necessary to reconstruct the full representation.

All these desiderata are satisfied by the networks in the RAAM family. Depending on the nature of the structures at hand and on how they are represented, different networks of the RAAM family can be used. A standard RAAM network is able to generate reduced descriptors for trees with

information stored in the leaves; an SRAAM network is, instead, suited for lists and an LRAAM network (which is a generalization of the SRAAM network) can deal with labeled directed graphs.

The general form of a network of the RAAM family is defined by a $N_I - N_H - N_O$ feed-forward encoder network, where both the input and output layers are partitioned in one label field and n reduced descriptor (pointer) fields, i.e., $N_I = N_O = N_l + nN_H$, where $N_l \geq 0$ is the size of the label field; moreover, the reduced descriptor fields and the hidden layer have the same size. In this formulation, the network for a RAAM network is obtained by setting $N_l = 0$ and the network for an SRAAM by setting $n = 1$. The set of equations describing the network are as follows:

$$\boldsymbol{F}_E(\vec{i}) = \boldsymbol{F}(E\vec{i} + \vec{\theta}_H) \equiv \vec{h}, \tag{1.3}$$

$$\boldsymbol{F}_D(\vec{h}) = \boldsymbol{F}(D\vec{h} + \vec{\theta}_O) \equiv \vec{o}, \tag{1.4}$$

where $\vec{h}, \vec{\theta}_H \in I\!R^{N_H}$, $\vec{o}, \vec{i}, \vec{\theta}_O \in I\!R^{N_l + nN_H}$, $\boldsymbol{F}(\vec{i})_j = f(\vec{i}_j)$, and $f()$ is a sigmoid-shaped function. The vectors $\vec{\theta}_H$ and $\vec{\theta}_O$ are the bias vectors for the hidden and output layers, respectively, $\vec{i}$ is the input pattern to the network, $\vec{h}$ is the hidden activation, and $\vec{o}$ is the output of the network. The matrix $E \in I\!R^{N_H \times N_I}$ is the Encoding matrix, i.e., the weight matrix between the input layer ($N_I = N_l + nN_H$ units) and the hidden layer (N_H units); and $D \in I\!R^{N_O \times N_H}$ is the Decoding matrix, i.e., the weight matrix between the hidden layer and the output layer ($N_O = N_I$). In the following, the index t will vary on the output units, s on the hidden units, and q on the input units.

A pattern[3] in the training set of a general RAAM network assumes the form $\vec{p} = [\vec{c}_1, \ldots, \vec{c}_m]$, where $\vec{c}_j \in I\!R^{N_H}$ can be either a reduced descriptor or a label. When considering an SRAAM network the typical form of a pattern becomes $\vec{p} = [\vec{l}, \vec{rd}]$, where $\vec{l} \in I\!R^{N_l}$ is a label and $\vec{rd} \in I\!R^{N_H}$ is a reduced descriptor. An LRAAM network generalizes this form, allowing multiple reduced descriptors: $\vec{p} = [\vec{l}, \vec{rd}_1, \ldots, \vec{rd}_k]$.

The learning algorithm for RAAM and SRAAM networks was originally developed by J. B. Pollack [Pol90] and it essentially remains the same for the LRAAM. The algorithm combines a standard gradient descent with a dynamical adaptation of the training set. Specifically, the goal of the gradient descent consists in minimizing the following cost function:

$$C_0 = \frac{1}{2} \sum_{k}^{P} \| \vec{p}^{\,(k)} - \boldsymbol{F}_D(\boldsymbol{F}_E(\vec{p}^{\,(k)})) \,|^2, \tag{1.5}$$

[3] For the sake of notation, in the following, we omit the transpose operator: the notation $\vec{p} = [\vec{y}_1, \ldots, \vec{y}_k]$ must be read as $\vec{p}^{\,t} = [\vec{y}_1^{\,t}, \ldots, \vec{y}_k^{\,t}]$.

where P is the number of patterns (substructures) in the training set.

If the learning is able to reach zero error, then the network implements a perfect compressor-reconstructor couple for the training set. Furthermore, if each reduced descriptor $\vec{rd}^{(k)}$ in the training set is equal to $\vec{h}^{(k)} = \boldsymbol{F}_E(\vec{p}^{(k)})$, then the network is also a member of the RAAM family, in the sense that each reduced descriptor retains all the information contained in the pattern $\vec{p}^{(k)}$. In fact, this information can be reconstructed by decoding $\vec{rd}^{(k)}$, i.e., $\vec{p}^{(k)} = \boldsymbol{F}_D(\vec{rd}^{(k)})$. Pollack's algorithm fulfills this constraint by modifying the training set during learning: at each epoch of learning every reduced descriptor $\vec{rd}^{(k)}$ is replaced by the corresponding hidden layer activity $\vec{h}^{(k)}$. Note that, this technique guarantees the consistency of the model only at global minima of C_0. Practically, however, even a point in the weights space which is in the proximity of a global minimum works well.

For LRAAM networks, this author introduced an additional learning rule for dealing with void descriptors [Spe94b]. This rule is necessary because in LRAAM networks there is no commitment to a particular representation for the void descriptor, i.e., the void descriptor can get multiple representations. These representations are distinguished from non-void descriptors thanks to a void condition bit allocated in the label for each reduced descriptor field of the network. The learning rule for void descriptors consists in copying the output representations for the void descriptor into the training set at each epoch of learning.

The main theoretical concern about Pollack's algorithm (and its LRAAM generalization) is whether it is possible to give it a better mathematical characterization, since an improved understanding of the underpinning mathematics should eventually either help in improving the efficiency of the algorithm or in disclosing the limitations of the RAAM family.

1.4 From Graph Representation to Graph Transductions

In the previous section, we have discussed a specific way of representing labeled graphs within a RAAM neural network. The basic idea is to implement a mechanism for encoding the graph into a vectorial representation, preserving all the information needed for a subsequent decoding of the vectorial representation in order to reconstruct the original graph. In this section, we discuss how the encoding function can be fruitfully combined

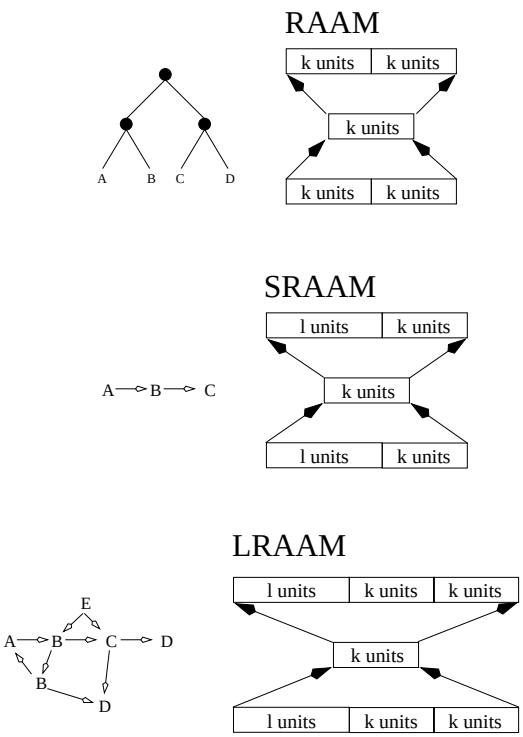


Figure 1.1: The models belonging to the RAAM family: the RAAM is able to encode trees with labels on the leaves, the SRAAM encodes sequences, while the LRAAM is able to encode labeled directed graphs.

with a transformation to obtain a specific type of graph transduction[4].

A functional graph transduction $\mathcal{T} : \mathcal{I} \rightarrow \mathcal{O}$ is defined as a mapping from an input structured domain $\mathcal{I}$ (over $\Sigma_{\mathcal{I}}$) to an output structured domain $\mathcal{O}$ (over $\Sigma_{\mathcal{O}}$). A special case of transduction that we will consider is the one where $\mathcal{O}$ is the trivial set of structures compounded by a single labeled node. In fact, it is not difficult to recognize that when this is the case, with $\Sigma_{\mathcal{O}} = \{0, 1\}$ the transduction can be interpreted as a classification function of the structures in $\mathcal{I}$.

In general, however, graph transductions can assume several forms. For the purpose of this tutorial we mainly focus on the class of transductions, defined on domains of DOAGS with supersource, which can be represented in the following form

$$\mathcal{T} = g \circ \hat{\tau} \tag{1.6}$$

where g is the *output* function and $\hat{\tau}$ is the *encoding* (or *state transition*) function. Specifically, $\hat{\tau}$ is defined recursively as

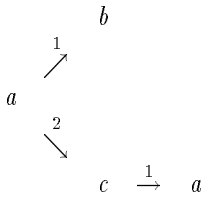
$$\hat{\tau}(\boldsymbol{Y}) = \begin{cases} nil & \text{if } \boldsymbol{Y} = \xi \\ \tau(s, \boldsymbol{Y}_s, \hat{\tau}(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}(\boldsymbol{Y}^{(o)})) & \text{otherwise} \end{cases} \tag{1.7}$$

where $\tau : vert(\mathcal{I}^{\#}) \times \mathcal{I} \times \mathcal{X}^o \rightarrow \mathcal{X}$ is the *c-model* function, $s = source(\boldsymbol{Y})$, $nil \in \mathcal{X}$ is used to represent the void graph, and $\boldsymbol{Y}^{(1)}, \ldots, \boldsymbol{Y}^{(o)}$ are the subgraphs pointed by s, i.e., $source(\boldsymbol{Y}^{(i)}) = ch_i[s]$. The function τ is called c-model function since it defines a computational model for the encoding function.

Note that, because of eq. (1.7), $\mathcal{T}$ is *causal* since τ only depends on the current node and nodes descending by it. Moreover, when τ does not depend on any specific vertex, i.e., $\tau(\boldsymbol{Y}_s, \hat{\tau}(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}(\boldsymbol{Y}^{(o)}))$, then $\mathcal{T}$ is also *stationary*. We will mainly focus on stationary transductions.

Of course, for general DOGs, the above definition of $\hat{\tau}(\cdot)$ is not adequate. In Section 1.8 we will also discuss how to deal with cyclic DOGs.

Example 1 *Given a stationary encoding function $\hat{\tau}$, the encoding of the labeled structure*



[4] Some experimental results obtained on combining an LRAAM network with a perceptron are reported in [SSG95].

is defined by the following set of equations

$$\hat{\tau}\left(\begin{array}{ccccc} & & b & & \\ & \overset{1}{\nearrow} & & & \\ a & & & & \\ & \underset{2}{\searrow} & & & \\ & & c & \overset{1}{\rightarrow} & a \end{array}\right) = \tau(a, \hat{\tau}(\mathbf{b}), \hat{\tau}(c \overset{1}{\rightarrow} a)),$$

$$\hat{\tau}(\mathbf{b}) = \tau(b, nil, nil),$$

$$\hat{\tau}(c \overset{1}{\rightarrow} a) = \tau(c, \hat{\tau}(\mathbf{a}), nil),$$

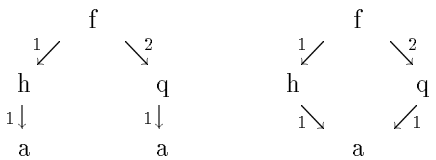
$$\hat{\tau}(\mathbf{a}) = \tau(a, nil, nil),$$

where **b** *and* **a** *denote the graphs with a single node labeled b, and a, respectively. By proper substitutions, the encoding can finally be written as*

$$\hat{\tau}\left(\begin{array}{ccccc} & & b & & \\ & \overset{1}{\nearrow} & & & \\ a & & & & \\ & \underset{2}{\searrow} & & & \\ & & c & \overset{1}{\rightarrow} & a \end{array}\right) =$$

$$= \tau(a, \tau(b, nil, nil), \tau(c, \tau(a, nil, nil), nil))$$

It should be noted that causal stationary transductions cannot compute any function of the input DOAG. For example, consider the following two DOAGs



Any given transduction described by equations (1.6-1.7) will necessarily map these two graphs into the same output, regardless of the form of functions $g(\cdot)$ and $\tau(\cdot)$. This is because the graph's processing is bottom-up, so it is not possible to recognize that in the right-side graph $pa[a]$ is constituted by two nodes, i.e., the ones labeled h and q.

This computational limitation, however, allows to improve the efficiency of computation. In fact, when considering DOAGs, the training set can be optimized by allowing only one occurrence of the same subgraph: if there

are graphs $\boldsymbol{X}^{(1)}, \ldots, \boldsymbol{X}^{(q)}$ which share a common subgraph $\hat{\boldsymbol{X}}$, then $\hat{\boldsymbol{X}}$ can be represented only once. This is obtained by merging all the graphs in a single minimal DOAG. After that, a topological sort on the vertices of the minimal DOAG is performed to determine the updating order on the vertices for the recursive network. The advantage of having this topological order is that all the reduced representations (and also their derivatives with respect to the weights) can be computed by a single ordered scan of the training set.

Both the merging[5] and sorting operations can be done efficiently (almost linearly) with respect to the size of all DOAGs and the size of the minimal DOAG, respectively. Moreover, the use of the minimal DOAG leads to a considerable reduction in space complexity. In some cases, such reduction can even be exponential.

1.5 Neural Graph Transductions

The implementation of graph transductions through standard and recurrent neural networks can only be realized by resorting to complex and very unnatural encoding protocols which map structures onto fixed-size unstructured patterns or sequences. A more natural approach, instead, can be obtained by proper instantiation of the input and output domains for g, $\hat{\tau}$, and τ.

Specifically, let $\mathcal{I}^{\#}$ be the input structured domain with real valued vectors as labels ($\mathcal{I} = I\!R^n$), and $\mathcal{O} = I\!R^k$. The encoding function $\hat{\tau}$ is completely defined by choosing a representation for the void graph and by defining the c-model function τ. By choosing $\mathcal{X} = I\!R^m$, any real valued vector would be fine for representing the void graph, however, for computational reasons which will be clear in the following, we choose the null vector for representing the void graph, i.e., $nil = \mathbf{o} \in I\!R^m$. Consequently the c-model function τ will be defined as

$$\tau : I\!R^n \times \underbrace{I\!R^m \times \cdots \times I\!R^m}_{\text{o times}} \rightarrow I\!R^m \tag{1.8}$$

where the first domain of the cartesian product denotes the label space, while the remaining domains represent the encoded subgraphs spaces up to the maximum outdegree of the input domain $\mathcal{I}^{\#}$. Finally, we can define the output function g as

$$g : I\!R^m \rightarrow I\!R^k. \tag{1.9}$$

[5] The merging operation can be performed by using either hash tables or AVL trees.

Note that eqs. (1.8) and (1.9) only describe the general form for τ and g. Different realizations can be given which satisfy the above equations. For example, both τ and g can be implemented by feedforward neural networks. Before to reach such level of complexity, however, it is worth to explore simpler realizations. Specifically, in the following section we study recursive neurons.

1.6 Recursive Neurons

Let first consider the case in which $m = 1$. The simplest non-linear neural realization for $\tau(\cdot)$ is given by

$$\tau(\boldsymbol{l}, x_1, \ldots, x_o) = f(\sum_{i=1}^{n} w_i l_i + \sum_{j=1}^{o} \hat{w}_j x_j + \theta), \tag{1.10}$$

where f is a sigmoidal function, w_i are the weights associated to the label space, $\hat{w}_j$ are the weights associated to the subgraphs spaces, θ is the bias, $\boldsymbol{l}$ is the current input label, and $x_1, \ldots, x_o$ are the encoded representations of subgraphs.

Example 2 (Logical term `f(a,g(b),c).`) *Let define a coding function*

$$\phi : \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{f}, \mathtt{g}\} \rightarrow \{0, 1\}^5$$

which encodes the symbolic labels into binary vectors, i.e., $\phi(\mathtt{a}) = [1, 0, 0, 0, 0]$, $\phi(\mathtt{b}) = [0, 1, 0, 0, 0]$, $\phi(\mathtt{c}) = [0, 0, 1, 0, 0]$, $\phi(\mathtt{f}) = [0, 0, 0, 1, 0]$, $\phi(\mathtt{g}) = [0, 0, 0, 0, 1]$, *and* $nil = 0$. *Thus,* $\tau(\cdot)$ *will be defined by five parameters for the label, i.e.,* $w_1, \ldots, w_5$, *three parameters for the subgraphs (maximum outdegree), i.e.,* $\hat{w}_1$, $\hat{w}_2$, *and* $\hat{w}_3$, *and the bias* θ.

Then,

$$\hat{\tau}\left(\begin{array}{ccc} & \mathtt{f} & \\ {\scriptstyle 1}\swarrow & {\scriptstyle 2}\downarrow & \searrow{\scriptstyle 3} \\ \mathtt{a} & \mathtt{g} & \mathtt{c} \\ & {\scriptstyle 1}\downarrow & \\ & \mathtt{b} & \end{array}\right) =$$

$$= f(\sum_{i=1}^{5} w_i \phi_i(\mathtt{f}) + \hat{w}_1 \overbrace{f(\sum_{i=1}^{5} w_i \phi_i(\mathtt{a}) + \theta)}^{\hat{\tau}(\mathbf{a})} +$$

$$\hat{w}_2\, f(\sum_{i=1}^{5} w_i\phi_i(\mathsf{g}) + \hat{w}_1\, \underbrace{f(\sum_{i=1}^{5} w_i\phi_i(\mathsf{b}) + \theta)}_{\hat{\tau}(\mathbf{b})} + \theta)$$

$$+\hat{w}_3\, \underbrace{f(\sum_{i=1}^{5} w_i\phi_i(\mathsf{c}) + \theta)}_{\hat{\tau}(\mathbf{c})} + \theta).$$

A graphical representation of the above equation is given in Figure 1.2.

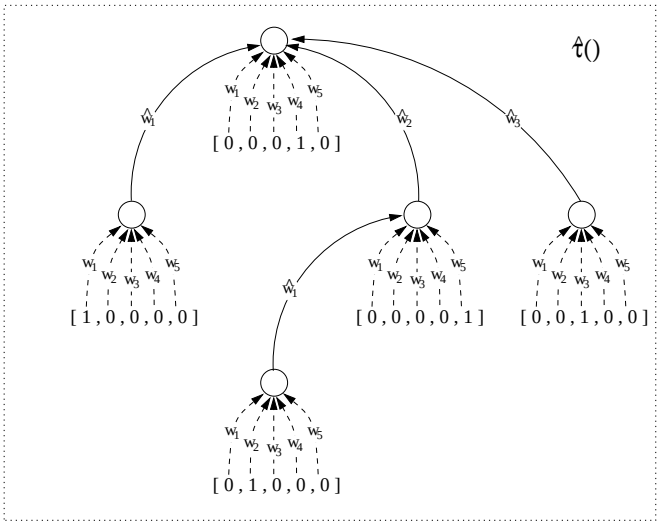


Figure 1.2: Graphical representation for the encoding function of Example 2.

When $m > 1$, $\tau(\cdot) \in I\!R^m$ can be written as

$$\tau(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) = \boldsymbol{F}(\boldsymbol{W}\boldsymbol{l} + \sum_{j=1}^{o} \widehat{\boldsymbol{W}}_j \boldsymbol{x}^{(j)} + \boldsymbol{\theta}), \tag{1.11}$$

where $\boldsymbol{F}_i(\boldsymbol{v}) = f(v_i)$, $\boldsymbol{l} \in I\!R^n$, $\boldsymbol{\theta} \in I\!R^m$, $\boldsymbol{W} \in I\!R^{m\times n}$, $\boldsymbol{x}^{(j)} \in I\!R^m$, $\widehat{\boldsymbol{W}}_j \in I\!R^{m\times m}$.

Concerning the output function $g(\cdot)$, it can be defined as a standard neuron taking in input the encoded representation of the graph, i.e.,

$$g(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{M}\boldsymbol{x} + \boldsymbol{\beta}), \tag{1.12}$$

where $\boldsymbol{M} \in \mathbb{R}^{k \times m}$ and $\boldsymbol{\beta} \in \mathbb{R}^k$ are the weight matrix and bias terms defining $g(\cdot)$, respectively.

As a special case, $g(\cdot)$ can be defined as a function which selects a subset of the state vector in $\mathbb{R}^m$. For example,

$$g(\boldsymbol{x}) = \boldsymbol{M}\boldsymbol{x}, \tag{1.13}$$

where $\boldsymbol{M} \in \mathbb{R}^{k \times m}$ is a 0-1 matrix with a single 1 for each row and at most a single 1 for each column. This corresponds to have the selected components of the state vector to contribute to both the output and the coding of the state.

The relationship between recurrent and recursive neurons is clarified by the following example.

Example 3 *A recurrent neuron can be considered as a special case of a recursive neuron applied to lists, i.e., labeled graphs with o=1; in that case, the position of a vertex within the list corresponds to the time of processing; e.g., given the list*

$$\boldsymbol{L} \equiv \boldsymbol{l}^{(k)} \rightarrow \boldsymbol{l}^{(k-1)} \rightarrow \ldots \rightarrow \boldsymbol{l}^{(1)}$$

the set of equations derived by the recursive application of eq. (1.10) is

$$x_1 = \tau(\boldsymbol{l}^{(1)}, nil) = f(\sum_{j=1}^{n} w_j l_j^{(1)})$$

$$x_i = \tau(\boldsymbol{l}^{(i)}, x_{i-1}) = f(\sum_{j=1}^{n} w_j l_j^{(i)} + \hat{w}_1 x_{i-1})) \qquad i = 2, \ldots, k$$

where $\hat{w}_1$ is the weight on the recursive connection; by making the time t of processing explicit, the above equations can be rewritten as the output $o(\cdot)$ of a recurrent neuron as follows

$$o(t=1) = f(\sum_{j=1}^{n} w_j l_j^{(1)})$$

$$o(t) = f(\sum_{j=1}^{n} w_j l_j^{(t)} + \hat{w}_1 o(t-1)) \qquad t = 2, \ldots, k.$$

1.7 Learning Algorithms

In this section, we discuss how several standard supervised algorithms for neural networks can be extended to structures.

1.7.1 Back-propagation through Structure

The task addressed by back-propagation through time networks [RM86] is to produce particular output sequences in response to specific input sequences. These networks are, generally, fully recurrent, in which any unit may be connected to any other. Consequently, they cannot be trained by using plain back-propagation. A trick, however, can be used to turn an arbitrary recurrent network into an equivalent feed-forward network when the input sequences have a maximum length T. In this case, all the units of the network can be duplicated T times (*unfolding of time*), so that the state of a unit in the recurrent network at time t is held by the tth copy of the same unit in the feed-forward network. By preserving the same weight values through the layers of the feed-forward network, it is not difficult to see that the two networks will behave identically for T time steps. The feed-forward network can be trained by back-propagation, taking care to preserve the identity constraint between weights of different layers. This can be guaranteed by adding together the individual gradient contributions of corresponding copies of the same weight and then changing all copies by the total amount.

Back-propagation through Time can easily be extended to structures. The basic idea is to use recursive neurons to encode the structures, i.e., $\hat{\tau}(\cdot)$; the representations obtained are then classified or used to approximate an unknown function by a standard feed-forward network, i.e., $g(\cdot)$.

Following our framework, given an input graph $\boldsymbol{X}$, the network output $\mathcal{T}(\boldsymbol{X})$ can be expressed as the composition of the encoding function $\hat{\tau}(\cdot)$ and the classification (or approximation) function $g(\cdot)$:

$$\mathcal{T}(\boldsymbol{X}) = g(\hat{\tau}(\boldsymbol{X})). \tag{1.14}$$

Learning for the set of weights $\boldsymbol{W}_g$ can be implemented by plain back-propagation on the feed-forward network realizing $g(\cdot)$

$$\Delta \boldsymbol{W}_g = -\eta \frac{\partial Error(g(\boldsymbol{y}))}{\partial \boldsymbol{W}_g}, \tag{1.15}$$

where $\boldsymbol{y} = \hat{\tau}(\boldsymbol{X})$, i.e., the input to the feed-forward network, while learning for the set of weights, $\boldsymbol{W}_{\hat{\tau}}$, realizing $\hat{\tau}(\cdot)$ can be implemented by

$$\Delta \boldsymbol{W}_{\hat{\tau}} = -\eta \frac{\partial Error(g(\boldsymbol{y}))}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}_{\hat{\tau}}}, \tag{1.16}$$

where the first term represents the error coming from the feed-forward network and the second one represents the error due to the encoding function $\hat{\tau}(\cdot)$. Note that $\boldsymbol{W}_{\hat{\tau}}$ is actually referring to the weights associated with the c-model function $\tau(\cdot)$. Thus, in order to clarify this point we will denote this weight matrix as $\boldsymbol{W}_{\tau}$

Goller and Küchler [GK96] devised a learning algorithm, namely *Back-propagation Through Structure*, on the basis of the above equations. The basic idea it to observe that $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}_{\tau}}$ can be computed by backpropagating the error from the feed-forward network implementing $g(\cdot)$ through the network implementing the encoding function $\hat{\tau}(\boldsymbol{X})$. As in back-propagation through time, the gradient contributions of corresponding copies of the same weight are collected for each structure. The total amount is then used to change all the copies of the same weight. If learning is performed by *structure*, then the weights are updated after the presentation of each individual structure, otherwise, the gradient contributions are collected through the whole training set and the weights changed after all the structures in the training set have been presented to the network. Recalling that the encoding network of a graph $\boldsymbol{X}$ is defined as the composition of the network implementing $\hat{\tau}(\boldsymbol{X})$ with the network implementing $g(\cdot)$, the learning algorithm for a (optimized) training set of L structures can be summarized in the following way:

Backpropagation Through Structure

Repeat

for l=1 **to** L

Compute $\mathcal{T}(\boldsymbol{X})$;

Compute the gradient using the Backpropagation Algorithm over the encoding network of $\boldsymbol{X}$;

Collect (add) the gradient for the different instances of weights in $\boldsymbol{W}_{\tau}$;

Update $\boldsymbol{W}_{\tau}$ and $\boldsymbol{W}_{g}$;

Until Convergence

1.7.2 Extension of Real-Time Recurrent Learning

The extension of Real-Time Recurrent Learning [WZ89] to recursive neurons does not present particular problems. Here we consider a neural model where $g(\cdot)$ is implemented as in equation (1.13), while the encoding is performed by m recursive neurons, i.e., (by including the bias into the label weight matrix)

$$\hat{\tau}(\boldsymbol{X}) = \tau(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) = \boldsymbol{F}(\boldsymbol{W}\boldsymbol{l} + \sum_{j=1}^{o} \widehat{\boldsymbol{W}}_j \boldsymbol{x}^{(j)}), \tag{1.17}$$

where $\boldsymbol{x}^{(j)} = \hat{\tau}(\boldsymbol{X}^{(j)})$ with $source(\boldsymbol{X}^{(j)}) = ch_j[source(\boldsymbol{X})]$, and

$$\boldsymbol{W}_\tau = [\boldsymbol{W}, \widehat{\boldsymbol{W}}_1, \ldots, \widehat{\boldsymbol{W}}_o]. \tag{1.18}$$

Here we show how to compute the derivatives of $\hat{\tau}(\boldsymbol{X})$ leaving to the reader the derivation of the final learning rule according to the chosen error function.

The derivatives of $\hat{\tau}(\boldsymbol{X})$ with respect to $\boldsymbol{W}$ and $\widehat{\boldsymbol{W}}_i$ ($i \in [1, \ldots, o]$) can be computed as follows:

$$\frac{\partial \hat{\tau}_p(\boldsymbol{X})}{\partial W_{tk}} = \tau_p'(\boldsymbol{l}_k \delta_{pt} + \sum_{j=1}^{o} (\widehat{\boldsymbol{W}}_j)_p \frac{\partial \boldsymbol{x}^{(j)}}{\partial W_{tk}}), \tag{1.19}$$

where τ_p' is the first derivative of the pth component of $\tau(\cdot)$ computed on the supersource of $\boldsymbol{X}$, δ_{mt} is the Kronecker delta, $p = 1, \ldots, m$, $t = 1, \ldots, m$, $k = 0, \ldots, n$ and $(\widehat{\boldsymbol{W}}_j)_p$ is the pth row of $\widehat{\boldsymbol{W}}_j$;

$$\frac{\partial \hat{\tau}_p(\boldsymbol{X})}{\partial (\widehat{\boldsymbol{W}}_i)_{tq}} = \tau_p'(\boldsymbol{x}_p^{(j)} \delta_{pt} + \sum_{j=1}^{o} (\widehat{\boldsymbol{W}}_j)_t \frac{\partial \boldsymbol{x}^{(j)}}{\partial (\widehat{\boldsymbol{W}}_i)_{tq}}), \tag{1.20}$$

where $t = 1, \ldots, m$, and $q = 1, \ldots, m$.

These equations are recursive on the structure $\boldsymbol{X}$, and can be computed by noting that if $\boldsymbol{V}$ is a graph composed by a single vertex, then

$$\frac{\partial \hat{\tau}_p(\boldsymbol{V})}{\partial W_{tk}} = \tau_p'(\boldsymbol{l}_{source(\boldsymbol{V})})_k \delta_{pt}, \text{ and } \frac{\partial \hat{\tau}_t(\boldsymbol{V})}{\partial (\widehat{\boldsymbol{W}}_i)_{tq}} = 0. \tag{1.21}$$

This allows the computation of the derivatives in real time alongside the computation of the neural representations for the graphs.

1.7.3 Recursive Cascade Correlation

In this section we discuss how a neural graph transduction $\mathcal{T}$ can be learned using an extension of the Cascade Correlation algorithm. The standard Cascade-Correlation algorithm [FL90] creates a neural network using an incremental approach for the classification (or regression) of unstructured patterns. The starting network $\mathcal{N}_0$ is a network without hidden nodes trained by a Least Mean Square algorithm. If network $\mathcal{N}_0$ is not able to solve the problem, a hidden unit u_1 is added such that the *correlation* between the output of the unit and the residual error of network $\mathcal{N}_0$ is maximized[6]. The weights of u_1 are frozen and the remaining weights are retrained. If the obtained network $\mathcal{N}_1$ cannot solve the problem, new hidden units are added which are connected (with frozen weights) with all the inputs and previously installed hidden units. The resulting network is a *cascade* of nodes. Fahlman extended the algorithm to the classification of sequences, obtaining good results [Fah91].

In the following, we show that the Cascade-Correlation can be further extended to structures by using our computational scheme [SMS96]. In fact, the shape of the c-model function can be expressed component-wise by the following set of equations:

$$\begin{aligned}
\tau_1 &= h_1(\boldsymbol{l}, \hat{\tau}_1(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}_1(\boldsymbol{Y}^{(o)})) && (1.22)\\
\tau_2 &= h_2(\boldsymbol{l}, \hat{\tau}_1(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}_1(\boldsymbol{Y}^{(o)}), && (1.23)\\
&\quad \hat{\tau}_2(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}_2(\boldsymbol{Y}^{(o)}), \hat{\tau}_1(\boldsymbol{Y}))\\
&\quad \vdots\\
\tau_m &= h_m(\boldsymbol{l}, \hat{\tau}_1(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}_1(\boldsymbol{Y}^{(o)}), && (1.24)\\
&\quad \hat{\tau}_2(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}_2(\boldsymbol{Y}^{(o)}), \ldots\\
&\quad \ldots, \hat{\tau}_m(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}_m(\boldsymbol{Y}^{(o)}), && (1.25)\\
&\quad \hat{\tau}_1(\boldsymbol{Y}), \ldots, \hat{\tau}_{m-1}(\boldsymbol{Y}))
\end{aligned}$$

where the h_i are suited nonlinear functions of the arguments.

Specifically, the output of the kth hidden unit, in our framework, can be computed as

$$\tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) = f(\sum_{i=0}^{n} w_i^{(k)} l_i + \sum_{v=1}^{k} \sum_{j=1}^{o} \hat{w}_{(v,j)}^{(k)} \boldsymbol{x}_v^{(j)} + \qquad (1.26)$$

[6] Since the maximization of the correlation is obtained using a gradient ascent technique on a surface with several maxima, a pool of hidden units is trained and the best one selected.

$$+\sum_{q=1}^{k-1} \bar{w}_q^{(k)} \tau_q(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})),$$

where $w_{(v,j)}^{(k)}$ is the weight of the kth hidden unit associated with the output of the vth hidden unit computed on the jth subgraph code $\boldsymbol{x}^{(j)}$, and $\bar{w}_q^{(k)}$ is the weight of the connection from the qth (frozen) hidden unit, $q < k$, and the kth hidden unit. The output of the network (with k inserted hidden units) is then computed according to equation (1.12).

Learning is performed as in standard Cascade Correlation by interleaving the minimization of the total error function (LMS) and the maximization of the correlation of the new inserted hidden unit with the residual error. The main difference with respect to standard Cascade Correlation is in the calculation of the derivatives. According to equation (1.26), the derivatives of $\tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})$ with respect to the weights are computed as

$$\frac{\partial \tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})}{\partial w_i^{(k)}} = f'(l_i + \sum_{j=1}^{o} \hat{w}_{(k,j)}^{(k)} \frac{\partial \boldsymbol{x}_k^{(j)}}{\partial w_i^{(k)}}) \tag{1.27}$$

$$\frac{\partial \tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})}{\partial \bar{w}_q^{(k)}} = f'(\tau_q(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) + \tag{1.28}$$

$$+\sum_{j=1}^{o} \hat{w}_{(k,j)}^{(k)} \frac{\partial \boldsymbol{x}_k^{(j)}}{\partial \bar{w}_q^{(k)}})$$

$$\frac{\partial \tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})}{\partial \hat{w}_{(v,t)}^{(k)}} = f'(\boldsymbol{x}_v^{(t)} + \sum_{j=1}^{o} \hat{w}_{(v,j)}^{(k)} \frac{\partial \boldsymbol{x}_k^{(j)}}{\partial \hat{w}_{(v,t)}^{(k)}}) \tag{1.29}$$

where $i = 0, \ldots, n$, $q = 1, \ldots, (k-1)$, $v = 1, \ldots, k$, $t = 1, \ldots, o$, f' is the derivative of $f()$. The above equations are recurrent on the structures and can be computed by observing that for graphs composed by a single vertex equation (1.27) reduces to $\frac{\partial \boldsymbol{x}_k^{(j)}}{\partial w_i^{(k)}} = l_i f'$, and all the remaining derivatives are null. Consequently, we only need to store the output values of the unit and its derivatives for each component of a structure.

Learning for the output weights proceeds as in the standard Cascade Correlation.

1.7.4 Extension of Neural Trees

Neural Trees (NT) have recently been proposed as a fast learning method in classification tasks. They are decision trees [BFOS84] where the splitting

of the data for each vertex, i.e., the classification of a pattern according to some features, is performed by a perceptron [SN90] or a more complex neural network [SM91]. After learning, each vertex at every level of the tree corresponds to an exclusive subset of the training data and the leaf nodes of the tree completely partition the training set. In the operative mode, the internal nodes route the input pattern to the appropriate leaf node which represents the class of it.

One advantage of the neural tree approach is that the tree structure is constructed dynamically during learning and not linked to a static structure like a standard feed-forward network. Moreover, it allows incremental learning, since subtrees can be added as well as deleted to recognize new classes of patterns or to improve generalization. Both supervised [SN90, Set90, Aa92, SM91] and unsupervised [PI92, LFJ92, Per92] splitting of the data have been proposed.

The learning and classification algorithms for binary and general trees can be found in [SM91]. The extension of these algorithms to structures is straightforward: the standard discriminator (or network) associated to each node of the tree is replaced by a recursive discriminator (or network) which can be trained with any of the learning algorithms we have presented so far.

1.8 Cyclic Graphs

Up to now we have considered DOAGs. In fact, the processing of graphs with cycles poses at least two main problems: *i)* how to modify the definition of $\hat{\tau}(\cdot)$ in such a way to have an unambiguous neural representation of the graph ? *ii)* how to modify the learning algorithm ? Of course, the answer to question *ii)* depends on the answer to question *i)*. Here we propose a solution to both problems, which however must be considered partial since it is not guaranteed to be valid under any condition. The nature of this limitation will be clear in the following.

Given a cyclic DOG $\boldsymbol{Y}$, with h vertices, let define the associated set of equations

$$\mathcal{S}(\boldsymbol{Y}) \quad = \quad \begin{cases} \mathbf{x}_{v_1} = \tau(v_1, \boldsymbol{Y}_{v_1}, \mathbf{x}_{ch_1[v_1]}, \ldots, \mathbf{x}_{ch_o[v_1]}) \\ \mathbf{x}_{v_2} = \tau(v_2, \boldsymbol{Y}_{v_2}, \mathbf{x}_{ch_1[v_2]}, \ldots, \mathbf{x}_{ch_o[v_2]}) \\ \vdots \\ \mathbf{x}_{v_h} = \tau(v_h, \boldsymbol{Y}_{v_h}, \mathbf{x}_{ch_1[v_h]}, \ldots, \mathbf{x}_{ch_o[v_h]}) \end{cases} \tag{1.30}$$

where there is one equation for each vertex $v_j \in vert(\boldsymbol{Y})$, and if $ch_i[v_j]$ is void, then $\mathbf{x}_{ch_i[v_j]} = nil$. Then, if $\mathcal{S}(\boldsymbol{Y})$ admits a (unique) solution, we

define

$$\hat{\tau}(\boldsymbol{Y}) = \mathbf{x}_{source(\boldsymbol{Y})}, \tag{1.31}$$

where we exploit the fact that, by definition, $source(\boldsymbol{Y})$ returns a predefined supersource of $\boldsymbol{Y}$ among all possible ones. Note that if $\boldsymbol{Y}$ is acyclic, then the system $\mathcal{S}(\boldsymbol{Y})$ has a unique solution given by eq. (1.7). So, eq. (1.31) is more general than eq. (1.7). The system $\boldsymbol{Y}$, however, in general may not allow a solution, or if it does, it may admit multiple solutions. Consequently, according to the way $\tau(\cdot)$ is actually implemented, sufficient conditions should be derived to guarantee the existence of a (unique) solution, thus solving problem *i)*. Such conditions, should then also be extended to cope with the perturbations introduced by the learning algorithm over the parametric implementation of $\tau(\cdot)$.

The neural implementation of eq. (1.31) can still be obtained by resorting to eqs. (1.10-1.12). In this case the encoding function $\hat{\tau}(\boldsymbol{Y})$ is computed by a recurrent neural network $\mathcal{N}(\boldsymbol{Y})$ defined according to $\mathcal{S}(\boldsymbol{Y})$: $\mathcal{N}(\boldsymbol{Y})$ will contain a pool of recursive neurons[7] for each distinct equation in $\mathcal{S}(\boldsymbol{Y})$ and the connections between pools of neurons is defined according to the functional dependencies expressed by each single equation in the system.

Example 4 *Let consider the following graph*

$$\begin{array}{ccccccccc}
\boldsymbol{Y} & = & \boldsymbol{l}^{(6)} & \xrightarrow{1} & \boldsymbol{l}^{(5)} & \xleftarrow{1} & \boldsymbol{l}^{(4)} & & \\
 & & & & {\scriptstyle 1}\downarrow & & {\scriptstyle 1}\uparrow & & \\
 & & & & \boldsymbol{l}^{(3)} & \xrightarrow{2} & \boldsymbol{l}^{(2)} & \xrightarrow{2} & \boldsymbol{l}^{(1)} \\
 & & & & {\scriptstyle 1}\downarrow & & & & \\
 & & & & \boldsymbol{l}^{(0)} & & & &
\end{array}$$

where $\boldsymbol{l}^{(0)} = [1,0,0,0,0]$, $\boldsymbol{l}^{(1)} = [0,1,0,0,0]$, $\boldsymbol{l}^{(2)} = [1,1,0,0,0]$, $\boldsymbol{l}^{(3)} = [0,0,1,0,0]$, $\boldsymbol{l}^{(4)} = [1,0,1,0,0]$, $\boldsymbol{l}^{(5)} = [0,0,0,1,0]$, $\boldsymbol{l}^{(6)} = [0,0,1,0,1]$. *Consider a single recursive neuron* ($m = 1$). *In this case,* $\tau(\cdot)$ *is defined by five parameters for the label, i.e.,* $w_1, \ldots, w_5$, *two parameters for the pointers,*

[7] The number of recursive units in the pool is given by m.

i.e., $\hat{w}_1$, and $\hat{w}_2$, and the bias θ. The corresponding system $\mathcal{S}(\boldsymbol{Y})$ is

$$\mathcal{S}(\boldsymbol{Y}) = \begin{cases} x_{v_0} = f(\sum_{j=1}^{5} w_j l_j^{(0)} + \theta) \\ x_{v_1} = f(\sum_{j=1}^{5} w_j l_j^{(1)} + \theta) \\ x_{v_2} = f(\sum_{j=1}^{5} w_j l_j^{(2)} + \hat{w}_1 x_{v_4} + \\ \qquad +\hat{w}_2 x_{v_1} + \theta) \\ x_{v_3} = f(\sum_{j=1}^{5} w_j l_j^{(3)} + \hat{w}_1 x_{v_0} + \\ \qquad +\hat{w}_2 x_{v_2} + \theta) \\ x_{v_4} = f(\sum_{j=1}^{5} w_j l_j^{(4)} + \hat{w}_1 x_{v_5} + \theta) \\ x_{v_5} = f(\sum_{j=1}^{5} w_j l_j^{(5)} + \hat{w}_1 x_{v_3} + \theta) \\ x_{v_6} = f(\sum_{j=1}^{5} w_j l_j^{(6)} + \hat{w}_1 x_{v_5} + \theta) \end{cases} \tag{1.32}$$

The resulting network $\mathcal{N}(\boldsymbol{Y})$ is shown in Figure (1.3).

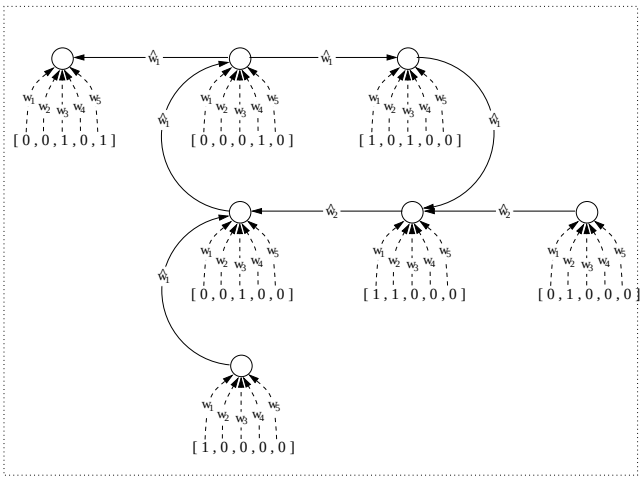


Figure 1.3: Graphical representation for the encoding function of the cyclic graph.

Satisfactory sufficient conditions to guarantee the existence of a unique solution to system $\mathcal{S}(\boldsymbol{Y})$ when using recursive neurons are still missing. In fact, it is well known that if the weight matrix of a recurrent network (as the network $\mathcal{N}(\boldsymbol{Y})$) is symmetric, an additive network with first order connections possesses a Liapunov function and is convergent ([Hop84, CG83]). Moreover, Almeida [Alm87] proved a more general symmetry condition than the symmetry of the weight matrix, i.e., a system satisfying *detailed balance* is guaranteed to possess a Liapunov function as well. Finally, if

the norm of the weight matrix (not necessarily symmetric) is sufficiently small, the network's dynamics can be shown to go to a unique equilibrium for a given input (Atiya [Ati88]). The above results, however, may impose too much constraints on the class of functions which can be computed and also they do not solve the stability problem for learning, i.e., the weights of the network $\mathcal{N}(\boldsymbol{Y})$ change with learning and there is no guarantee that learning will preserve the stability of the network.

1.9 Learning with Cycles

In this section we formulate a proposal on how learning in the case of cyclic graphs can be performed. Both extensions of Back-propagation and Real-Time learning are discussed.

1.9.1 Back-propagation

Let consider the framework introduced in Section 1.7.1. Since the network implementing $\tilde{\tau}(\cdot)$ is recurrent, $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}_\tau}$ can only be computed by resorting to *recurrent back-propagation*[Pin88], which has been defined to cope with neural networks containing cycles.

In the standard formulation, a recurrent back-propagation network $\mathcal{N}$ with p units is defined as

$$\boldsymbol{o}^{(rbp)}(t+1) = \boldsymbol{F}(\boldsymbol{W}^{(rbp)}\boldsymbol{o}^{(rbp)}(t) + \boldsymbol{I}^{(rbp)}), \tag{1.33}$$

where $\boldsymbol{I}^{(rbp)} \in I\!R^p$ is the input vector for the network, and $\boldsymbol{W}^{(rbp)} \in I\!R^p \times I\!R^p$ the weight matrix. The learning rule for a weight of the network is given by

$$\Delta w_{rs}^{(rbp)} = \eta o_s^{(rbp)} o_r'^{(rbp)} \sum_k e_k (\boldsymbol{L}^{-1})_{kr} \tag{1.34}$$

where all the quantities are taken at a fixed point of the recurrent network, e_k is the error between the current output of unit k and the desired one, $L_{ji} = \delta_{ji} - o_j'^{(rbp)} w_{ji}$ (δ_{ji} is a Kronecker delta), and the quantity

$$y_j^{(rbp)} = \sum_k e_k (\boldsymbol{L}^{-1})_{kj} = \sum_i o_i'^{(rbp)} w_{ij} y_i^{(rbp)} \ + \ e_i \tag{1.35}$$

can be computed by relaxing the *adjoint* network $\mathcal{N}'$, i.e., a network obtained from $\mathcal{N}$ by reversing the direction of the connections[8]. The weight

[8] Note that, in this case, the directions of the connections agree with the directions of the corresponding edges in the input graph.

w_{ji} from neuron i to neuron j in $\mathcal{N}$ is replaced by $o_i^{(rbp)} w_{ij}$ from neuron j to neuron i in $\mathcal{N}'$. The activation functions of $\mathcal{N}'$ are linear and the output units in $\mathcal{N}$ become input units in $\mathcal{N}'$ with e_i as input.

Given a cyclic graph $\boldsymbol{Y}$, let $h = \#vert(\boldsymbol{Y})$, m be the number of recursive neurons and $\boldsymbol{o}_i(t)$ be the output at time t of these recursive neurons for $v_i \in vert(\boldsymbol{Y})$. We define the global state vector of $\mathcal{N}(\boldsymbol{Y})$ at time t as

$$\boldsymbol{o}(t) = \begin{bmatrix} \boldsymbol{o}_1(t) \\ \boldsymbol{o}_2(t) \\ \vdots \\ \boldsymbol{o}_h(t) \end{bmatrix} \tag{1.36}$$

where, for convention, $\boldsymbol{o}_1(t)$ is the output of the neurons representing $source(\boldsymbol{Y})$.

To account for the labels, we have to slightly modify eq. (1.33)

$$\boldsymbol{o}(t+1) = \boldsymbol{F}(\widehat{\boldsymbol{W}}_\tau \boldsymbol{o}(t) + \boldsymbol{W}_\tau \boldsymbol{I}), \tag{1.37}$$

where

$$\boldsymbol{I} = \begin{bmatrix} \boldsymbol{l}_1 \\ \vdots \\ \boldsymbol{l}_h \end{bmatrix} \tag{1.38}$$

is the vector collecting all the labels of $\boldsymbol{Y}$, and

$$\boldsymbol{W}_\tau = \underbrace{\begin{bmatrix} \boldsymbol{W} & & 0 \\ & \ddots & \\ 0 & & \boldsymbol{W} \end{bmatrix}}_{h \text{ repetitions of } \boldsymbol{W}} \tag{1.39}$$

and $\widehat{\boldsymbol{W}}_\tau \in \Re^{hm \times hm}$ is defined according to the topology of $\boldsymbol{Y}$.

The input of the adjoint network is $e_k = (\frac{\partial Error(g(\boldsymbol{y}))}{\partial \boldsymbol{y}})_k$, and the learning rules become:

$$\Delta \hat{w}_{rs} = \eta o_s o'_r \sum_k e_k (\boldsymbol{L}^{-1})_{kr}, \tag{1.40}$$

$$\Delta w_{rs} = \eta I_s o'_r \sum_k e_k (\boldsymbol{L}^{-1})_{kr}. \tag{1.41}$$

Of course, as in Back-propagation Through Structure, all the changes referring to the same weight are added and then all copies of the same weight

are changed by the total amount. It must be noted that each graph gives rise to a new adjoint network and independently contributes to the variations of the weights. Moreover, the above formulation is more general than Back-propagation Through Structure: when used for DOAGs the adjoint network becomes a feedforward network representing the back-propagation of errors.

As for standard recurrent networks, also in this case it may be necessary to force the creation of new fixed points in the dynamics of the network by using *teacher forcing*, a variation of the recurrent Back-propagation proposed by Pineda [Pin88]. However, while the effects of this technique is quite well understood in standard recurrent networks, it is not clear to us how it can influence the dynamics of the networks associated to the input graphs. In fact, given a set of graphs $\{\boldsymbol{Y}^{(1)}, \ldots, \boldsymbol{Y}^{(p)}\}$, the associated networks $\mathcal{N}(\boldsymbol{Y}^{(1)}), \ldots, \mathcal{N}(\boldsymbol{Y}^{(p)})$ are interdependent since they share the same set of weights. A more accurate study on how this sharing of resources affects the dynamics of the system when using teacher forcing is needed.

1.9.2 Real-Time

Before proceeding with definitions and equations, we need to clarify a basic assumption inherent into the application of real-time learning to cyclic graphs. In fact, the original real-time algorithm was defined for sequences, where the order of presentation of the input data is trivially defined. This order can naturally be extended when dealing with DOAGs: a node is presented only if all its offsprings have been presented. Of course, the application of this convention does not work for cyclic graphs since the presence of a cycle traversing all the vertex of the graph will render impractical the application of a real-time algorithm: the encoding of the graph can be computed only at the time all the vertices of the graph are presented and the corresponding network is relaxed. Thus, in this case, it is more efficient to use the extension of Back-propagation presented in the previous section. Consequently, in the following we only consider graphs with local cycles, i.e., cycles involving few vertices. In this case, it is useful to resort to two strictly related concepts, the strongly connected components of a graph and the related definition of a component graph.

Definition 1 (Strongly Connected Component) *A strongly connected component of a directed graph $\boldsymbol{G}$ is a maximal set of vertices $U \subseteq \mathrm{vert}(\boldsymbol{G})$ such that for every pair of vertices u and v in U, there exists a path from u to v and vice-versa, i.e., u and v are reachable from each other.*

Definition 2 (Component Graph) *The (acyclic) component graph of $\boldsymbol{G}$ is defined as the graph $\boldsymbol{G}^{SCC}$, where $vert(\boldsymbol{G}^{SCC})$ contains one vertex*

for each strongly connected component of $\boldsymbol{G}$*, and* $edg(\boldsymbol{G}^{SCC})$ *contains the edge* (u, v) *if there is a directed edge from a vertex in the strongly connected component of* $\boldsymbol{G}$ *corresponding to* u *to a vertex in the strongly connected component of* $\boldsymbol{G}$ *corresponding to* v*.*

The computation of the strongly connected components of a graph $\boldsymbol{G}$ and its component graph $\boldsymbol{G}^{SCC}$ can be performed by combining two depth-first searches in $\Theta(\#vert(\boldsymbol{G}) + \#edg(\boldsymbol{G}))$ in time [CLR90].

The basic assumption is that vertices of input graphs are presented to the recursive network according to the preorder defined by the corresponding component graphs. Once that the vertices belonging to a strongly connected component are presented, the partial encoding of the corresponding subgraph can be computed alongside with the relevant derivatives. It is in this sense that we perform a real-time processing.

Thus, the computation of the derivatives in real-time for a graph $\boldsymbol{Y}$ can be done by observing that

1. equations 1.19 and 1.20 are still valid;

2. each strongly connected component of $\boldsymbol{Y}$ corresponds to a set of interdependent equations of this kind;

3. $\boldsymbol{Y}^{SCC}$ defines the dependences between sets of equations.

Example 5 *Let consider a single recursive neuron* ($m = 1$) *and the graph shown in Example 4. The component graph* $\mathbf{Y}^{SCC}$ *is given by*

$$u_3 \rightarrow u_2 \rightarrow u_1$$
$$\downarrow$$
$$u_0$$

where

$$u_0 \stackrel{\text{def}}{=} \boldsymbol{l}^{(0)} \qquad\qquad \boldsymbol{l}^{(5)} \leftarrow \boldsymbol{l}^{(4)}$$
$$u_1 \stackrel{\text{def}}{=} \boldsymbol{l}^{(1)} \qquad u_2 \stackrel{\text{def}}{=} \downarrow \qquad \uparrow$$
$$u_3 \stackrel{\text{def}}{=} \boldsymbol{l}^{(6)} \qquad\qquad \boldsymbol{l}^{(3)} \rightarrow \boldsymbol{l}^{(2)}$$

Let us consider the derivatives $x'_{v_j} = \frac{\partial x_{v_j}}{\partial \hat{w}_1}$ *with* $v_j \in vert(\boldsymbol{Y})$. *They are the solutions of the set of equations (derived by equation 1.20)*

$$\begin{cases} component\ u_0 \\ x'_{v_0} = 0 \\ \\ component\ u_1 \\ x'_{v_1} = 0 \\ \\ component\ u_2 \\ x'_{v_2} = f'(x_{v_4} + \hat{w}_1 x'_{v_4} + \hat{w}_2 x'_{v_1}) \\ x'_{v_3} = f'(x_{v_0} + \hat{w}_1 x'_{v_0} + \hat{w}_2 x'_{v_2}) \\ x'_{v_4} = f'(x_{v_5} + \hat{w}_1 x'_{v_5}) \\ x'_{v_5} = f'(x_{v_3} + \hat{w}_1 x'_{v_3}) \\ \\ component\ u_3 \\ x'_{v_6} = f'(x_{v_5} + \hat{w}_1 x'_{v_5}) \end{cases}$$

which can be decomposed according to the component graph. Specifically, by substituting the solution for x'_{v_1} *into the equation for* x'_{v_2}, *and the solution for* x'_{v_0} *into the equation for* x'_{v_3}, *the subset of equations corresponding to the strongly connected component* u_2 *can be solved through a relaxation process. Finally, once the solution value for* x'_{v_5} *is known, the last equation corresponding to component* u_3 *can be solved by direct substitution. In general, by taking into account the dependences established by the component graph, both the output of the recursive neuron and its derivatives can be computed in real-time with respect to the component graph.*

1.10 Computational Power

It is well known that recurrent neural networks can simulate any finite state automata [ADO91, OG96] as well as any multi-stack Turing machine in real time [SS95]. When constraining the network architecture, however, this computational power may no longer hold. For example, Elman-style Recurrent Networks can simulate any finite state automata [GGCC94, Kre95], while Recurrent Cascade-Correlation cannot [GCS+95, Kre96]. For this reason, it is of paramount importance to assess the computational power of a given network architecture, since this characterizes the class of functions which, in principle, can be computed by such network. Given an application domain, and based on the observation that the difficulty of training a network is directly proportional to the computational power exhibited by

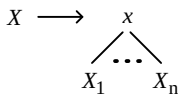
the network, computational results can be used to select the least complex architecture able to deal with the application.

To assess the computational capability of recursive neural networks, it turns useful to resort to Tree Automata theory. In the next section we give a brief introduction to Tree Grammars and to a specific class of Tree Automata, i.e., Frontier-to-Root Tree Automata.

1.10.1 Tree Grammars and Tree Automata

A *tree grammar* is defined as a four-tuple $G_t = (V, r, P, S)$ where $V = N \cup \Sigma$ is the grammar alphabet (nonterminals and terminals); (V, r) a ranked alphabet; productions in P are of the form $T_i \rightarrow T_j$, where T_i and T_j are trees; and S in T_V is a finite set of "starting trees," where T_V denotes the set of trees with nodes labeled by elements in V.

A tree grammar is in *expansive form* if all its productions are of the form



(1.42)

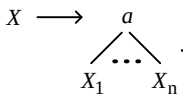
A (deterministic) *frontier-to-root tree automaton* (FRA) is a system $\mathcal{A}_t = (Q, F, \{f_a | a \text{ in } \Sigma\})$ where Σ is a ranked alphabet; Q is a finite set of states; F is a set of final states, a subset of Q; $\{f_a | a \text{ in } \Sigma\}$ is a set of transition functions $f_a : Q^m \rightarrow Q$ such that m is a rank of the symbol $a \in \Sigma$. The recognition process computed by the automaton can be described intuitively by the following computational stages: first of all, the frontier of the state tree is labeled q_0 (the status associated to nodes with $m = 0$, i.e, the leaves); then, for any node in the input labeled l with rank m, the corresponding node of the state tree is labeled $f_l(q_1, \ldots, q_m)$, where $q_1, \ldots, q_m$ are the states labeling the offsprings of that state tree node. Finally, an input tree is accepted by $\mathcal{A}_t$ if the automaton can enter a final state upon encountering the root.

According to the above definition, the *language recognized* by $\mathcal{A}_t$ is the set

$$T(\mathcal{A}_t) = \{T | T \text{ in } T_\Sigma,\ \mathcal{A}_t \text{ can halt in a state in } F \text{ when the root of } T \text{ is reached } \}.$$

It is not difficult to realize that, given an expansive tree grammar $G_t(V, r, S, P)$ generating the set $L(G_t)$ of trees with nodes labeled with elements in Σ, it is always possible to construct a FRA $\mathcal{A}_t$ that recognizes $L(G_t)$. In fact,

let $Q = N$ with $F = \{S\}$ and, for each symbol a in Σ, define a transition function f_a such that $f_a(X_1, \ldots, X_n) = X$ *iff* there is in G_t a production



(1.43)

1.10.2 Computational Results

All the results reported in this section are based on the observation that a sigmoid function can approximate a step function to an arbitrary degree of precision by augmenting the modulus of the associated weight vector. Thus, if it is demonstrated that a recursive network with step functions can implement any FRA, the result holds for recursive networks with sigmoids as well.

Elman-style networks, i.e., recursive networks with fully interconnected hidden units, turns out to be powerful enough to simulate any FRA [Spe97]:

Theorem 1.10.1 *An Elman-style network can simulate any FRA.*

The proof is based on a constructive procedure using step function units: there will be an hidden unit for each transition function of the FRA and a state is represented by the set of units associated to transition functions which introduce the same state in the computation.

Unfortunately, not all the neural networks for the processing of structures are as powerful as Elman-style networks. In fact, the following theorems states that both Cascade-Correlation networks and Neural Trees cannot simulate any FRA [Spe97].

Theorem 1.10.2 *A Cascade-Correlation network for structures cannot simulate any FRA.*

The proof for this theorem is obtained by observing that a Cascade-Correlation network for structures is a generalization of a Recurrent Cascade-Correlation network which has been proved unable to simulate any finite state machine [GCS+95, Kre96]. Since a finite state machine is equivalent to a FRA with relations having rank 1, it follows that a Cascade-Correlation network for structures, which in the finite state machine case reduces to a Recurrent Cascade-Correlation network, cannot simulate any FRA.

The same kind of limitation is proved for Neural Trees.

Theorem 1.10.3 *A Neural Tree for structures cannot simulate any FRA.*

This is demonstrated by showing the following result [Spe97]:

Theorem 1.10.4 *Any Neural Tree for structures can be implemented by a Cascade-Correlation network for structures.*

The above theorem holds since it is observed that a Neural Tree can be cast in the form of a three layered feedforward network. The trick is to retain all the units associated with internal nodes of the Neural Tree as units of the first layer. The second layer of units is then used to represent all the possible paths from the root to one leaf. Finally, the output layer will implement the **or** of the paths associated to leaves of the corresponding class. The proof is then concluded by observing that the feedforward network belongs to the class of architectures implemented by the Cascade-Correlation for structures.

Note that from the previous two Theorems, and from the fact that Cascade-Correlation networks and Neural Trees for structures are generalizations of Recurrent Cascade-Correlation networks and Neural Trees for sequences, respectively, we have

Corollary 1 *A Neural Tree for sequences cannot simulate any Finite State Machine.*

and

Corollary 2 *Any Neural Tree for sequences can be simulated by a Recurrent Cascade-Correlation network.*

A summary of the computational results is given in graphical form in Figure 1.4.

1.10.3 Function Approximation

Theoretical results on the capacity of recursive neural network to approximate an arbitrary mapping from a set of labeled trees to the reals have been developed in [HS97]. Specifically, given

- a set $\mathcal{S}$ of trees with labels in a finite set;
- a function $\bar{f} : \mathcal{S} \rightarrow I\!R^q$;
- set of functions $\mathcal{F}$ implementable by a recursive neural network defined as in [GK96];

the following result hold

Theorem 1.10.5 *For any $\epsilon > 0$ there exists a function $f^\epsilon \in \mathcal{F}$ such that $\forall s \in \mathcal{S},\ P(|f^\epsilon(s) - \bar{f}(s)| > \epsilon) < \epsilon$*

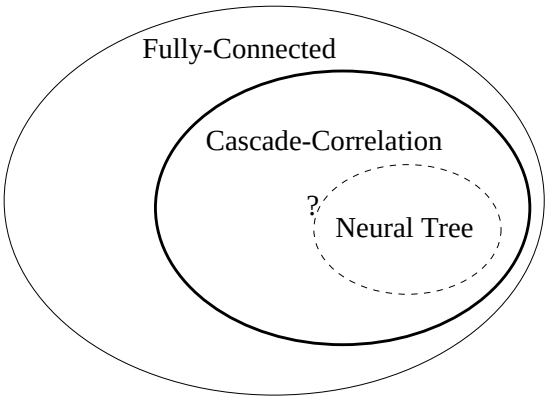


Figure 1.4: Graphical representation of the computational relationships between different recursive neural network architectures.

This result is quite interesting, since it states that if the set of possible labels is finite, then any mapping from trees to the set of reals can be implemented by a recursive neural network having a sufficient number of parameters. No universal mapping result for arbitrary real valued vector as labels, however, has been established up to now.

1.11 Complexity Issues

A usual problem in neural networks is the evaluation of how many resources a network needs to implement the desired function. In this section, we explore this problem from a node complexity perspective when considering the implementation of *Frontier-to-Root Tree Automata with Output* (FRAO) into recursive neural networks.

Here we report upper bounds to the number of units needed to implement a given FRAO in four, three, and two layers recursive networks. While the bound for four layers networks constitutes a generalization of the bound discussed in [HH96] for the implementation of FSM in recurrent networks, the bound for three layers networks produces, as a special case, a new bound to the implementation of FSA in three layers networks. Moreover, the bound for three layers networks is constructive and, thus, it may turn useful for practical applications where a priori knowledge available in form of formal languages or inferred by symbolic machine learning approaches can be injected into the neural network (for the case of recurrent neural networks see Omlin and Giles [OG96], Frasconi et al. [FGS95]). Its constructive proof is based on the encoding of each possible input configura-

tion to the FRA by a different integer in a predefined interval. This integer is then processed by the repeated application of the *telescopic* technique [SRK95], which is called in this way because it consists in the progressive activation of a set of units according to the magnitude of the processed integer. On the basis of this progressive encoding, it is possible to implement both the transition and output function of the desired FRAO.

Finally, a lower bound based on a counting argument is reported, which demonstrate the optimality of the upper bound obtained for the four layers recursive networks.

1.11.1 Representing FRAO as Boolean Functions

In [HH96] the m states of a FSA are encoded through $\log m$ bits. Hence, the state and output transition functions can be implemented by a boolean function

$$\{0,1\}^{\log m+\log l} \rightarrow \{0,1\}^{\log m+\log l} \tag{1.44}$$

where $\log m$ bits in input are used to encode the current status, while the remaining $\log l$ bit are used to encode the current input label. Similarly, $\log m$ bits in output are used to encode the new status and one bit to encode the output bit.

In a similar way an FRAO can be implemented by a boolean function

$$\{0,1\}^{\log l+N\log m} \rightarrow \{0,1\}^{\log l+\log m} \tag{1.45}$$

where $\log l$ is the number of bits encoding the input-output symbols, and N is the maximum rank for the symbols.

1.11.2 Upper Bounds

Even in this case, the fact that a sigmoid function can approximate a step function to an arbitrary degree of precision by augmenting the modulus of the associated weight vector, implies that any complexity bound based on threshold gates is valid also for sigmoidal networks.

It is known that Finite State Automata with m states and binary alphabet can be inserted in four layers recurrent neural networks with $O(\sqrt{m})$ units [HH96]. This result was obtained by exploiting the following lemma by Lupanov [Lup72]:

Lemma 1 *Arbitrary boolean logic functions of the form $f : \{0,1\}^x \rightarrow \{0,1\}^y$ can be implemented in a four layer network of perceptrons with a node complexity of $O(\sqrt{\frac{y2^x}{x-\log y}})$.*

Using the same lemma the following theorem can be proved:

Theorem 1.11.6 (Four Layers Insertion) *Any FRAO with m states, l input-output labels, and maximum rank N, can be implemented by a four layers recursive neural network with a node complexity of $O(\sqrt{\frac{(\log l+\log m)lm^N}{\log l+N\log m}})$.*

If $l < m$ the node complexity can be simplified to $O(\sqrt{\frac{lm^N}{N}})$.

If the network is constrained to only have three layers, then by using the telescopic technique [SRK95] it can be proven that

Theorem 1.11.7 (Three Layers Insertion) *Any FRAO with m states, l input-output labels, and maximum rank N, can be implemented by a three layers recursive neural network with a node complexity of $O((\log l + \log m)\sqrt{lm^N})$ and a number of connections of $O((\log l + \log m)\, lm^N)$.*

Note that the result stated in the theorem gives a direct upper bound to the insertion of FSA in three layers recurrent networks as well:

Corollary 3 *A three layers recurrent neural network can implement FSM having m states, and l input-output labels, with a node complexity of $O((\log l+\log m)\sqrt{lm})$ and a number of connections of $O((\log l + \log m)lm)$.*

Finally, when considering two layers networks, the application of the telescopic technique allows to establish the following theorem

Theorem 1.11.8 (Two Layers Implementation) *Any FRAO with m states, l input-output labels, and maximum rank N, can be implemented by a two layers recursive neural network with a node complexity of $O(lm^N)$ and $O((\log l + N\log m)lm^N)$ connections.*

1.11.3 A Lower Bound on the Node Complexity

When considering the minimum number of neurons that are required to implement arbitrary FRAO into recursive networks, a first naive consideration would lead to $\Omega(\log l + \log m)$ since l different input symbols and m states can effectively be encoded by $\log l + \log m$ units.

However, by using counting arguments similar in nature to the ones used by Alon et al. [ADO91] and Horn & Hush [HH96] (lower bounds on the node complexity for recurrent network implementations of FSA), it is possible to to derive a better lower bound:

Theorem 1.11.9 (Lower Bound, Node Complexity) *The node complexity required to implement an arbitrary FRAO with m states, l input-output labels and maximum rank N in a recursive neural network is $\Omega(\sqrt{\frac{(\log l+\log m)lm^N}{\log l+N\log m}})$.*

This result clearly states the optimality of the upper bound for the four layers networks.

1.11.4 Bounds on Learning

It is well known that any neural network learning algorithm may be seriously plagued by the presence of local minima in the associated error function. This is also true for the class of algorithms we have introduced here. Recently, however, the efficiency of learning the membership of DOAGs in terms of local minima of the error surface by relying on the principle that their absence is a guarantee of efficient learning as been analyzed [FGS97]. The authors give a sufficient condition under which the error surface is local minima free. In particular, they define a topological index associated with a collection of DOAGs that make it possible to design the architecture so as to avoid local minima. The condition they give holds for any training set composed of graphs with symbolic nodes and a neural network capable of learning the assigned data.

Finally, it is worth to mention that some preliminary results on the VC-dimension of a specific type of recursive neural network are given in [Ham97].

1.12 Conclusion

The possibility to represent and to process structures in a neural network greatly increases the potential of integration between neural networks and symbolic systems in a knowledge-based neurocomputing system. In fact, structures generated by a symbolic module can be evaluated by this type of networks and their evaluation can be used to modify the behavior of the symbolic module. An instance of this integration scheme is given, for example, by learning heuristics for automated deduction systems. Goller reported very successful results in using a Back-propagation Through Structure network within the SETHEO theorem prover [Gol97]. On the other hand, it is not difficult to figure out, in analogy with finite state automata extraction from recurrent networks, how to extract tree automata from a neural network for structures. This would allow the above scheme to work on the other side around, with a neural module which is driven by a symbolic subsystem.

Another application domain where neurocomputing of structures already showed promising results is in chemistry [BMSS]. In this domain, compounds are naturally represented as labeled graphs. However, the main computational target typically consists in assigning numerical ratings to

chemical compounds in order to establish their usefulness in specific applications. Neural networks for structures are particularly good in performing this kind of tasks.

From the above examples it should be clear that neural networks for structures have the potential to build a very natural bridge between two totally different ways of representing concepts. This constitutes a contribution towards the development of a homogeneous and systematic methodology for the integration of components of different type in a knowledge-based system.

Bibliography

[Aa92] L. Atlas and al. A performance comparison of trained multilayer perceptrons and trained classification trees. *Proceedings of the IEEE*, 78:1614–1619, 1992.

[ADO91] N. Alon, A.K. Dewdney, and Teunis J. Ott. Efficient Simulation of Finite Automata by Neural Nets. *Journal of the ACM*, 38(2):495–514, April 1991.

[Alm87] L.B. Almeida. A learning rule for asynchronous perceptrions with feedback in a combinatorial environment. In M. Caudil and C. Butler, editors, *Proceedings of the IEEE First Annual International Conference on Neural Networks*, pages 609–618. CA: IEEE, 1987.

[Ati88] A. Atiya. Learning on a general network. In D.Z. Anderson, editor, *Neural Information Processing Systems*, pages 22–30. New York: AIP, 1988.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth International Group, 1984.

[BMSS] A.M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Application of cascade correlation networks for structures to chemistry. *Applied Intelligence.* To appear.

[CG83] M. A. Cohen and S. Grossberg. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Trans. on Systems, Man, and Cybernetics*, 13:815–826, 1983.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[Fah91] S. E. Fahlman. The recurrent cascade-correlation architecture. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 190–196, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[FGS95] P. Frasconi, M Gori, and G. Soda. Recurrent Neural Networks and Prior Knowledge for Sequence Processing: A Constrained Nondeterministic Approach. *Knowledge-Based Systems*, 8(6):313–332, 1995.

[FGS97] P. Frasconi, M. Gori, and A. Sperduti. On the efficient classification of data structures by neural networks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1066–171, 1997.

[FGS98] P. Frasconi, M. Gori, and A. Sperduti. A framework for adaptive data structures processing. *IEEE Transactions on Neural Networks*, 9(5):768–786, 1998.

[FL90] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 524–532. San Mateo, CA: Morgan Kaufmann, 1990.

[GCS+95] C.L. Giles, D. Chen, G.Z. Sun, H.H. Chen, Y.C. Lee, and M.W. Goudreau. Constructive learning of recurrent neural networks: Limitations of recurrent casade correlation and a simple solution. *IEEE Transactions on Neural Networks*, 6(4):829–836, 1995.

[GGCC94] M.W. Goudreau, C.L. Giles, S.T. Chakradhar, and D. Chen. First-order vs. second-order single layer recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(3):511–513, 1994.

[GK96] C. Goller and A. Küchler. Learning task-dependent distributed structure-representations by backpropagation through structure. In *IEEE International Conference on Neural Networks*, pages 347–352, 1996.

[Gol97] Christoph Goller. *A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems.* Phd thesis, Technical University Munich, Faculty of Computer Science, 1997.

[GT78] R. C. Gonzalez and M. G. Thomason. *Syntactical Pattern Recognition*. Addison-Wesley, 1978.

[Ham97] B. Hammer. Learning recursive data is intractable. Osnabrücker Schriften zur Mathematik Reihe P, Heft 194, Fachbereich Mathematik/Informatik, Universität Osnabrück, 49069 Osnabrück, November 1997.

[Hay94] S. Haykin. *Neural Networks: a comprehensive Foundation*. Piscataway, NJ: IEEE Press, 1994.

[HH96] B. G. Horne and D. R. Hush. Bounds on the complexity of recurrent neural network implementations of finite state machines. *Neural Networks*, 9(2):243–252, 1996.

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

[HK91] L. H. Hall and L. B. Kier. *Reviews in Computational Chemistry*, chapter 9, The Molecular Connectivity Chi Indexes and Kappa Shape Indexes in Structure-Property Modeling, pages 367–422. VCH Publishers, Inc.: New York, 1991.

[Hop84] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. In *Proc. Natl. Acad. Sci.*, pages 3088–3092, 1984.

[HS97] B. Hammer and V. Sperschneider. Neural networks can approximate mappings on structured objects. In *ICCIN '97, 2nd International Conference on Computational Intelligence and Neuroscience*, 1997.

[Kre95] S.C. Kremer. On the computational power of elman-style recurrent networks. *IEEE Transactions on Neural Networks*, 6(4):1000–1004, 1995.

[Kre96] S.C. Kremer. Finite state automata that recurrent cascade-correlation cannot represent. In D. Touretzky, M. Mozer, and M. Hasselno, editors, *Advances in Neural Information Processing Systems 7*. MIT Press, 1996. 612-618.

[LFJ92] T. Li, L. Fang, and A. Jennings. Structurally adaptive self-organizing neural trees. In *International Joint Conference on Neural Networks*, pages 329–334, 1992.

[Lup72] O. Lupanov. Circuits using Threshold elements. *Soviet Physics – Doklady*, 17(2):91–93, 1972.

[Met91] E.J. Metcalfe. Recognition failure and charm. *Psychological Review*, 98(4):529–553, 1991.

[Mur93] B.B. Murdock. Todam2: A model for the storage and retrieval of item, associative, and serial-order information. *Psychological Review*, 100(2):183–203, 1993.

[OG96] C.W. Omlin and C.L. Giles. Constructing Deterministic Finite-State Automata in Recurrent Neural Networks. *Journal of the ACM*, 43(2):937–972, 1996.

[Per92] M. P. Perrone. A soft-competitive splitting rule for adaptive tree- structured neural networks. In *International Joint Conference on Neural Networks*, pages 689–693, 1992.

[PI92] M. P. Perrone and N. Intrator. Unsupervised splitting rules for neural tree classifiers. In *International Joint Conference on Neural Networks*, pages 820–825, 1992.

[Pin88] F. J. Pineda. Dynamics and architecture for neural computation. *Journal of Complexity*, 4:216–245, 1988.

[Pla95] T. A. Plate. Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6(3):623–641, 1995.

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

[RM86] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, 1986.

[Set90] I. K. Sethi. Entropy nets: From decision trees to neural networks. *Proceeding of the IEEE*, 78:1605–1613, 1990.

[SM91] A. Sankar and R. Mammone. *Neural Tree Networks*, pages 281–302. Neural Networks: Theory and Applications. Academic Press, 1991.

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

[SMS96] A. Sperduti, D. Majidi, and A. Starita. Extended cascade-correlation for syntactic and structural pattern recognition. In Petra Perner, Patrick Wang, and Azriel Rosenfeld, editors, *Advances in Structural and Syntactical Pattern Recognition*, Lecture notes in Computer Science, pages 90–99. Springer-Verlag, Berlin, 1996.

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

[Spe94a] 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.

[Spe94b] A. Sperduti. Labeling RAAM. *Connection Science*, 6(4):429–459, 1994.

[Spe97] A. Sperduti. On the computational power of recurrent neural networks for structures. *Neural Networks*, 10(3), 1997.

[SRK95] K.-Y. Siu, V. Roychowdhury, and T. Kailath. *Discrete Neural Computation*. Englewood Cliffs, New Jersey: Prentice Hall, 1995.

[SS95] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

[SS96] A. Sperduti and A. Starita. A general learning framework for the raam family. In Maria Marinaro and Roberto Tagliaferri, editors, *Neural Nets WIRN Vietri-95*, pages 136–141. World Scientific, Singapore, 1996.

[SS97] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[SSG95] A. Sperduti, A. Starita, and C. Goller. Learning distributed representations for the classification of terms. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 509–515, 1995.

[TH88] D. S. Touretzky and G.E. Hinton. A distributed connectionist production system. *Cognitive science*, 12(3):423–466, 1988.

[Tha73] J. W. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

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

[WZ89] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.