An Introduction to Learning Structured Information

Paolo Frasconi

Dipartimento di Sistemi e Informatica
Università di Firenze (Italy)
paolo@mcculloch.ing.unifi.it

1 Introduction

By and large, connectionist models have been successfully employed for solving learning tasks characterized by relatively poor data types. For example, feedforward neural networks and probabilistic mixture models can only deal with *static* data types such as records or fixed-size numerical arrays. Sequences are the first significant improvement over static data. There are two important new issues that arise when modelling data using sequential types. First, a sequence is a dynamic set of atomic entities, where each atom contains information stored as a static object, but the number of atoms is not fixed. This permits to represent data objects having variable size. Second, a *serial order* relation is defined among atoms, providing us with some additional information that is not encoded into the variables themselves. Serial order makes sequences naturally suited for modelling data in temporal domains, where atoms in a sequence can be associated with temporal events. However, serial order is only a very special relation, and more complex relations amongst atomic entities may exist in other learning domains. These relations can be conveniently represented using graphs. Examples of domains in which instances have a rich structure are quite numerous. For example, data in multimedia applications or in hearth sciences have temporal and spatial dimensions, generalizing sequences to regular multidimensional grids. Compounds in chemistry and molecular biology are naturally represented by undirected graphs. Complex graphical structures (such as labeled trees and webs) are very common in syntactic pattern recognition. Other domains such as automated reasoning, software engineering or the World Wide Web also yield instances which are represented by directed graphs. Sequences, from this point of view, should be regarded as the very special class of graphs whose topology is restricted to linear chains.

As largely described elsewhere in this book, connectionist models for sequence processing are based on the same recursive state updating scheme that characterizes the state space representation used in system theory to describe nonlinear dynamical systems. Recurrent neural networks (RNNs), hidden Markov models (HMMs) [1] and input-output HMMs [2] are very similar from this representational point of view, since they share the same recursive state updating scheme. Essentially, they can be seen as generalizations of their static counterparts, i.e. feedforward neural networks and probabilistic mixture models, respectively.

C. Lee Giles and Marco Gori (Eds.): Adaptive Processing, LNAI 1387, pp. 99–120, 1998.

The processing framework employed in this chapter is basically a generalization of the recursive state updating scheme, from sequences to directed acyclic graphs. While in sequential dynamics the state at a given time point t is a function of the state at the previous time point $t-1$, in graphical dynamics the state at a given vertex v is a function of the states at the children of that vertex. Graphical dynamics of this kind have been introduced several years ago in theoretical computer science, extending finite automata for strings to finite automata for trees or other labeled graphs [3]. During the Seventies, the interest in graphical automata was particularly boosted by potential applications to syntactical pattern recognition. The first connectionist architectures for data structures were only conceived with unsupervised learning problems, and were based on recursive autoassociative memories (RAAMs) [4] and labeling RAAMs (LRAAMs) [5, 6]. Only very recently the supervised learning problem has been taken into account, leading to generalize *recurrent* neural networks for sequences to *recursive* neural networks for data structures [7]. Extensions of hidden Markov models [1] to graphical data are also possible, although still very much unexplored.

In this chapter we introduce a graphical computational framework for learning data structures, but little emphasis is given to specific models and learning algorithms. The interested reader is advised to consult [8, 7] for a thorough review of recursive neural networks, and [9] for a more detailed exposition of some of the topics being covered in this chapter.

2 Two Examples of Structured Learning Domains

2.1 Pattern Recognition

Many of the early approaches to pattern recognition were based on syntactic approaches [10–13]. The method is essentially based on two steps. First, patterns are converted into symbolic structures, such as strings or labeled graphs. Then, a formal *grammar* is determined, in order to describe the rules according to which patterns should be classified. In this way, pattern classification is reduced to the problem of computing language membership. Syntactic techniques are extremely powerful since they can deal in a very natural way with complex structured domains. Unfortunately, devising a proper grammar is often very difficult task because domain knowledge is incomplete or insufficient to account for the several sources of uncertainty that typically affect the real world. This motivates research in the field of grammatical inference, where one is interested in learning syntactical rules from data [14, 15]. Soft adaptive models, such as neural networks and probabilistic graphical models, typically exhibit a high degree of robustness with respect to noise and missing or uncertain information. Therefore, they currently represent an important alternative paradigm for pattern recognition. Moreover, neural networks are known to be capable of robustly representing finite automata [16, 17] and interesting links have been established between symbolic and connectionist grammatical inference procedures [18]. Unfortunately, as discussed in the introduction, neural network models have been mostly specialized on rather trivial data types. Hence, pattern recognition may

take advantage of generalized neural networks for data structures by exploiting the best assets of both syntactic and connectionist approaches.

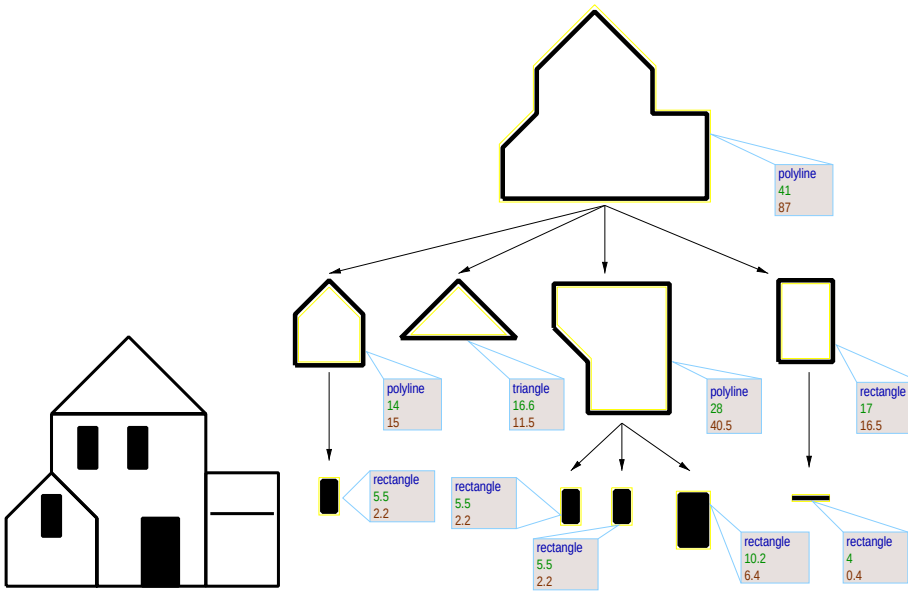


Fig. 1. A logo with the corresponding representation based on both symbolic and subsymbolic information.

Let us consider the problem of business logo recognition as a simple example for illustrating how pattern recognition may receive advantage of structured representations. Figure 1 shows an image representing a business logo. The most straightforward representation, often used by neural networks people, would simply be obtained by converting the image into a fixed-size array of numerical features. However, much of the information concerning the structure of the image (e.g., relative position of image elements) would be lost. An alternative, as shown in Fig. 1 is to represent the logo as a tree, whose vertices are associated with image components, properly described in terms of geometrical features. In the example, shape, perimeter, and area are measured for each image component. This representation is invariant with respect to roto-translations and naturally incorporate both symbolic and numerical information.

2.2 Automated Reasoning

Many interesting problems in artificial intelligence are solved by searching. Theorem provers, deductive databases, and symbolic expert systems all rely on inference engines that search a solution (or a proof) in a typically very complex

space. The search space is often represented as a tree, whose vertices are associated with search states, and edges represent inference steps. The ubiquitous difficulty in this kind of problems is the combinatorial explosion of the search space, that means computational intractability or even non computability. In order to avoid exhaustive search, AI systems often employ heuristic evaluation functions that estimate the cost associated with the exploration of a given part of the search tree. Evaluation functions make it possible to select the most promising branches of the search tree, thus permitting a considerable saving of time. For example, in the A* [19] and IDA* (iteratively deepening A*) algorithms, [20], the search tree is traversed depth-first, the successors of a node are rated by an evaluation function, and the successor with the best scoring is expanded next.

Unfortunately, heuristics are very expensive to be devised since they typically summarize the knowledge of an expert of a given domain. Moreover, they are too specific, so that even a slight change in the domain requires a new heuristic to be devised from scratch. An interesting method to avoid these difficulties is to learn heuristics from examples of solutions already found in other ways. The input domain for these learning tasks corresponds to the space of search states (i.e. the contents of vertices in the search tree) and desired outputs can be obtained by inspecting the search subtree rooted at the vertex being considered. A very simple approach would consist of labeling each search state as positive or negative, depending on whether or not a solution is present in the corresponding subtree. In some domains, the search state can be satisfactorily represented in a static form (e.g., in solving the famous 15 puzzle search states are just board configurations, that can be represented as an array of 16 integers) and a feedforward neural network could perform the learning task. In problems such as automated reasoning, however, search states are expressions in first-order logic and thus they need highly structured representations[1] [21]. In these cases, neural networks for data structures are naturally suited for learning search evaluation functions.

Terms in first order logic can be easily represented as a directed acyclic graphs, as shown in Fig. 2. Here vertices are labeled by function names and edges are used to link functions to their arguments. Constants (like α, β, γ) are considered to be functions with zero arity and, therefore, are always found on leaf vertices. An interesting application of recursive neural networks for learning heuristics in the context of automated deduction systems is described in [22].

3 Graphical Representation of Data

3.1 Notation

A graph used for describing structured information is a pair $G = (V, E)$, where V is a set of vertices and E is a binary relation on V. An edge $(v, w) \in E$ is

[1] Although in this case the search tree and the content of its vertices are both data structures, they should not be mixed up. The input to the learner is a representation of the current search state and not the whole search tree.

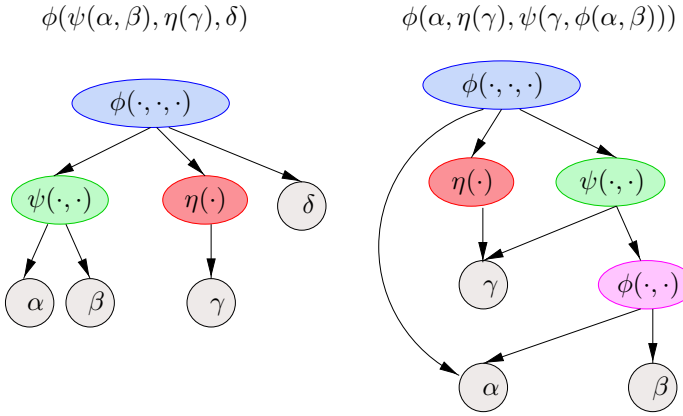


Fig. 2. A directed acyclic graph representing the logical term $\phi(\alpha, \psi(\gamma), \psi(\gamma, \phi(\alpha, \beta)))$.

undirected if both (v, w) and (w, v) are in E. Otherwise, if (w, v) is not in E, then (v, w) is directed. A graph is directed if all its edges are directed. The *undirected version* G' of a directed graph G is obtained by adding an edge (w, v) whenever the edge (v, w) exists in E. Two vertices v and w are *neighbors* if $(v, w) \in E$ or $(w, v) \in E$. An n-path from v to w is a sequence of vertices $v = v_0, \ldots, v_n = w$ such that $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots n$. An n-path is a cycle if $v = w$. A graph is acyclic if it has no cycles. If G is directed, then the *undirected paths* and the *undirected cycles* of G are the paths and the cycles in G'. The acronym DAG will be used when referring a directed acyclic graph.

The following definitions apply to directed graphs. If an edge (v, w) is present in E, then v is a *parent* of w and w is a *child* of v. Vertex w is a *descendant* of vertex v if there exists a path from v to w. In such a case, v is an *ancestor* of w. 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. The *outdegree* of vertex v is the number of edges (v, w) leaving from v and the *indegree* of vertex v is the number of edges incident on v. A vertex having zero indegree is a *root*. A vertex with zero indegree is a *leaf*. Vertex s is said to be a *supersource* for G if for each vertex in $V \setminus \{s\}$ there exist a path from s to v. If G is a DAG, then G has at most one supersource.

3.2 Label Spaces

A data structure is a graph whose nodes and edges are marked by sets of domain variables, called *labels*. We assume that all the labels in a graph are disjoint sets. The domain variables contained into labels are also called *attributes*, and may be either numerical (i.e., they take on continuous values) or categorical (i.e., they take on discrete, or symbolic values). The presence of an edge (v, w) in a marked graph indicates that the variables contained in v and w are related is some way.

For the sake of simplicity, we shall assume that edges are unlabeled. At any rate, edge labels in a directed graph can be easily moved into the labels attached to the vertex from which the edges emanate.

In the following we assume that an additional equivalence relation is defined among domain variables, where variables within an equivalence class have the same type and the same semantics. We say that two labels are *similar* if they contains at most one element from each equivalence class and intercept the same subset of equivalence classes. A graph is *uniformly labeled* if all its labels are similar. For example, in our logo example shape, area and perimeter are equivalence classes and the graph in Fig. 1 is uniformly labeled.

3.3 Skeleton Classes and Spaces of Data Structures

The skeleton of a data structure is obtained by ignoring all the labels, but retaining the topology of the graph. A class of skeletons is a set of unlabeled graphs that satisfy some specified topological conditions. For example, the class of DAGs is formed by all directed acyclic graphs. A generic class of skeletons is denoted by the symbol $\#$. Once we have specified a label space $\mathcal{Y}$ and a skeleton class $\#$, we may define the space of data structures with labels in $\mathcal{Y}$ and topology in $\#$. Such as space is denoted as $\mathcal{Y}^{\#}$. For example, let us consider a very special case. When $\mathcal{Y}$ is a finite alphabet and $\#$ is the class of linear chains (i.e., sequences), $\mathcal{Y}^{\#}$ corresponds to the set of all possible strings on the alphabet $\mathcal{Y}$. This set is known as the Kleene closure of $\mathcal{Y}$ in language theory. Spaces of data structures are based on a similar concept, but labels are not limited to symbols and structures are not limited to linear chains. In the following we shortly define some useful classes of skeletons.

Ordered Graphs A graph is *ordered* if a total order $\prec$ is defined on the edges leaving from each vertex. The class of DOAGs is formed by all directed acyclic ordered graphs (e.g., see Fig. 3). Ordering children turns out to be particularly useful for the recursive computational scheme that will be introduced later in this chapter. In some applications (e.g., graphs that represent logical terms), children are inherently ordered. In other cases children ordering can be decided according to some conventions. For example, in logo recognition, sub-images can be conventionally ordered by examining the global image the top to the bottom and from left to right.

Bounded DOAGs Other useful skeleton classes are obtained by imposing upper bounds on the the maximum indegree and the maximum outdegree of each vertex. The class of bounded-indegree graphs is defined as $\#_p = \{(V, E) : \forall v \in V \text{indegree}(v) \leq p\}$. The class of bounded-outdegree graphs is defined as $\#^m = \{(V, E) : \forall v \in V \text{outdegree}(v) \leq m\}$. The class of bounded-neighborhood graphs is defined as $\#_p^m = \#_p \cap \#^m$. When considering the class of bounded-outdegree DOAGs, or m-DOAGs, we conventionally fill in missing children using *external vertices*, as shown in Fig. 4. External vertices are unlabeled. The set of external vertices is called *frontier*.

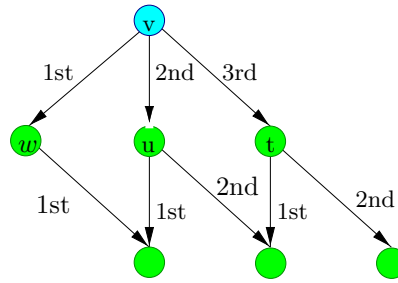


Fig. 3. A directed acyclic ordered graph. In this example $(v, w) \prec (v, u) \prec (v, t)$.

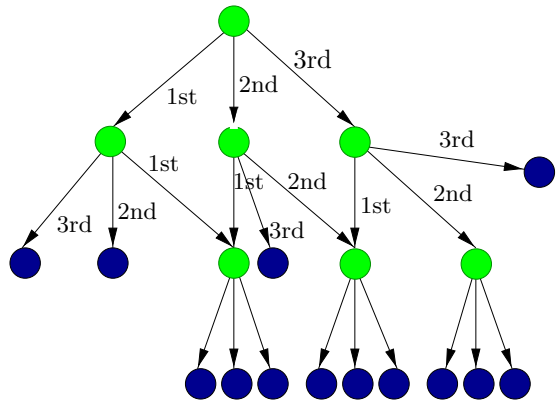


Fig. 4. The DOAG of Fig. 3 regarded as a bounded-outdegree DOAG with maximum outdegree 3. Missing children have been filled in by external vertices, shown as black vertices.

Positional Graphs An useful superclass of DOAGs is the class of directed *positional* acyclic graphs (DPAGs), in which it is assumed that for each vertex v, a bijection $P : E \rightarrow \mathbb{N}$ is defined on the edges leaving from v. DPAGs are a superclass of DOAGs in the sense that every DOAG is also a DPAG, but not viceversa. Bounded DPAGs, or m-DPAGs, are defined by restricting the range of P to the finite set $[1, m] \subset \mathbb{N}$, for some fixed m.

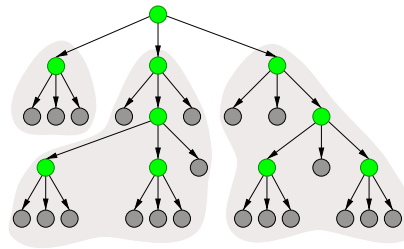


Fig. 5. A ternary tree.

An interesting subclass of m-DPAGs is formed by prohibiting cycles in the undirected version of the graph. Such as m-DPAGs are a quite common structure in computing and are known as m-ary trees. Often, m-ary trees are defined using mathematical induction, since this is the best way to highlight the recursive nature of the structure. Inductively, an m-ary tree is defined as being either

- an external vertex, or
- an ordered $m + 1$ tuple $(r, t_1, \ldots, t_m)$ where the *root* r is a vertex and t_i are m-ary trees for $i = 1, \ldots, m$.

As a very special case, when $m = 1$, m-ary trees reduce to linear chains, or lists. In the present framework, linear chains should be regarded as the graphical representation of sequences. In fact, the relation that corresponds to linear chains is a serial order. It is worth to remark that edges in a linear chain are conventionally directed in the opposite direction with respect to time, i.e. time indexes that may be thought to be associated with vertices decrease following the arrows.

Grids The last class of skeletons we considered is formed by multidimensional grids. Grids are particularly interesting for describing images or spatio-temporal domains, where atomic entities are indexed by a tuple of coordinates (e.g., x,y,z,and t). Clearly m-grids are a special case of m-DPAGs where the topology is constrained to a regular multidimensional pattern. When $m = 1$, grids clearly reduce to sequences.

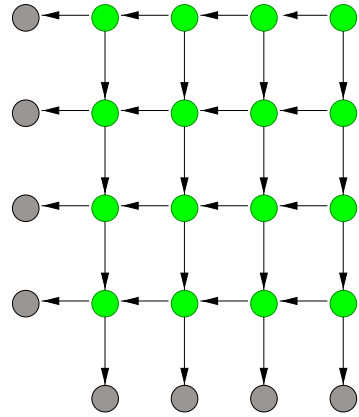


Fig. 6. A two-dimensional grid.

4 Structural Transductions

In the most general formulation, a deterministic transduction for structured domains is a relation $\tau \subset \mathcal{U}^{\#} \times \mathcal{Y}^{\#'}$, where $\mathcal{U}$ and $\mathcal{Y}$ are assigned input and output label spaces, respectively, and $\#$, $\#'$ are assigned skeleton classes. A probabilistic transduction is the joint density $P(\boldsymbol{Y}, \boldsymbol{U})$ of random data structures whose realizations belong to the Cartesian product $\mathcal{U}^{\#} \times \mathcal{Y}^{\#'}$. Learning general transductions defined in this way is an open research problem, and, to the best of our knowledge, no approach for solving it has been even suggested. Therefore, we shall limit ourselves to a very limited case of transduction and we shall assume the following hypotheses:

1. $\# \equiv \#'$ and τ is a function from $\mathcal{U}^{\#}$ to $\mathcal{Y}^{\#}$.
2. the skeleton of data structures in invariant under the transduction τ, i.e., for all $\boldsymbol{U} \in \mathcal{U}^{\#}$ we have $\text{skel}(\tau(\boldsymbol{U})) = \text{skel}(\boldsymbol{U})$. Transductions of this type are said to be *IO-isomorph*.
3. $\#$ is contained into the class of bounded DPAGs.

Notice that in the temporal domain, Hypothesis 2 simply means that the transduction being considered is *synchronous*, i.e. a single output Y_t is emitted in response to each single input U_t, at each time point[2]. The probabilistic counterpart of deterministic transductions which are functions from $\mathcal{U}^{\#}$ to $\mathcal{Y}^{\#}$ are conditional densities $P(\boldsymbol{Y}|\boldsymbol{U})$ of random data structures, where realizations of $\boldsymbol{Y}$ belong to $\mathcal{Y}^{\#}$ and realizations of $\boldsymbol{U}$ belong to $\mathcal{U}^{\#}$. For the sake of simplicity, in the following we shall mostly consider deterministic transductions. This is actually a special case since a (deterministic) function can be always seen as

[2] Note that even for sequences, learning asynchronous transductions is non a trivial problem. Symbolic solutions have been given in [23] and connectionist approaches have been reported in [24] and [25].

a particular conditional density which is zero almost everywhere. However, the concepts introduced in the following of this section are more easily explained in the deterministic case and the stochastic extensions are relatively straightforward.

4.1 Algebraic and Causal Transductions

A transduction $\tau(\cdot)$ is said to be *algebraic* or *unstructured* if $\forall \boldsymbol{U} \in \mathcal{U}^{\#}$ and $\forall v \in \text{vert}(\boldsymbol{U})$, $\tau(\boldsymbol{U})_v$ only depends on $\boldsymbol{U}_v$. Clearly, algebraic transductions can be simply realized by using a static function independently applied to each input label to obtain the corresponding output label and, therefore, they have no interest for us.

Causality is a concept often encountered in dynamical system theory, where it is normally defined by considering transductions that operate on sequential spaces. In particular, let $\tau : \mathcal{U}^{\#} \rightarrow \mathcal{Y}^{\#}$ a synchronous transduction Then τ is said to be causal if the output Y_t does not depend on the future inputs $U_{t+1}, U_{t+2}, \ldots$. Causality can be generalized to IO-isomorph transductions for structured domains. We shall say that $\tau(\cdot) : \mathcal{U}^{\#} \rightarrow \mathcal{Y}^{\#}$ is causal if the output label at node v only depends on the input substructure induced by v and its descendant. Causality for sequences is clearly a special case, as shown in Fig. 7.

4.2 Recursive Representations

Let us first consider the case of transductions defined on sequences. In this case, we say that a transduction $\tau()$ admits a recursive representation if, at any given time step t, there exists a variable $X_t \in \mathcal{X}$, called *state*, such that

$$X_t = f(X_{t-1}, U_t, t) \tag{1}$$

$$Y_t = g(X_t, U_t, t) \tag{2}$$

where $f : \mathcal{X} \times \mathcal{U} \times \mathbb{N} \rightarrow \mathcal{X}$ is called *state transition function* and $g : \mathcal{X} \times \mathcal{U} \times \mathbb{N} \rightarrow \mathcal{Y}$ is called *output function*. A recursive representation can only exist if the transduction is causal. In that case, the state X_t can be thought of a summary of the information contained in past inputs, which is sufficient in order to produce future outputs. More precisely, all the information about the subsequence $U_1, \ldots, U_{t-1}$ is not needed for determining $Y_t, \ldots, Y_T$, provided that X_t is known (see top of Fig. 8). The base step of recursion 1 is resolved by using the *initial state* X_0, which is assumed to be given[3]. Note that the base step corresponds to the (only) external vertex of the linear chain that graphically represents the input sequence.

We can now generalize recursive representations to bounded DPAGs. We say that an IO-isomorph transduction $\tau() : \mathcal{U}^{\#} \rightarrow \mathcal{Y}^{\#}$ admits a recursive representation if, for each node v in the skeleton of the input structure $\boldsymbol{U}$ there exists a

[3] In the case of recurrent neural networks, it has been shown that the initial state of the model can actually be learnt from data [26].

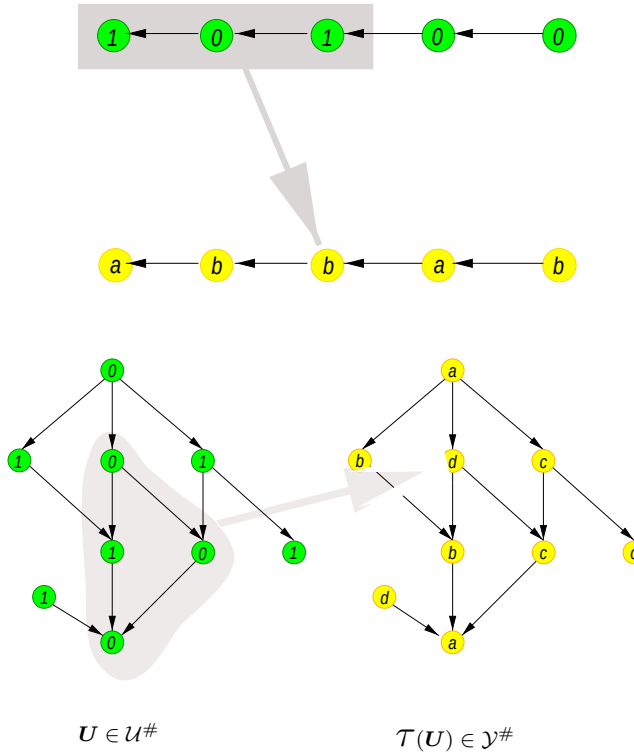


Fig. 7. Top: Causality in temporal domains. Bottom: Causality in structured domains.

state variable $X_v \in \mathcal{X}$ such that

$$X_v = f(\boldsymbol{X}_{\mathrm{ch}[v]}, U_v, v) \tag{3}$$

$$Y_v = g(X_v, U_v, v) \tag{4}$$

where $f : \mathcal{X}^m \times \mathcal{U} \times I\!N \to \mathcal{X}$ is the state transition function and $g : \mathcal{X} \times \mathcal{U} \times I\!N \to \mathcal{Y}$ is the output function. $\boldsymbol{X}_{\mathrm{ch}[v]}$ in equation 3 denotes the m-tuple of state variables attached to the children of vertex v. Whenever a child is missing, the corresponding entry in the tuple is replaced by a predetermined state $X_0 \in \mathcal{X}$, called *frontier state*. The frontier state plays the same role of the initial state for temporal transductions and it is associated with the base step of recursion (i.e., the external vertices). An obvious difference is that in the case of sequences there is only one external vertex.

Like in the case of sequences, a recursive representation can only exist if the transduction is causal. Here, the state X_v can be thought of a summary of the information contained in the input subgraph induced by v and its descendants. More precisely, all the information about the input subgraph induced by $\{v\} \cup \mathrm{de}[v]$ is irrelevant for determining Y_w, for each $w \in V \setminus \mathrm{de}[v]$, provided that X_v

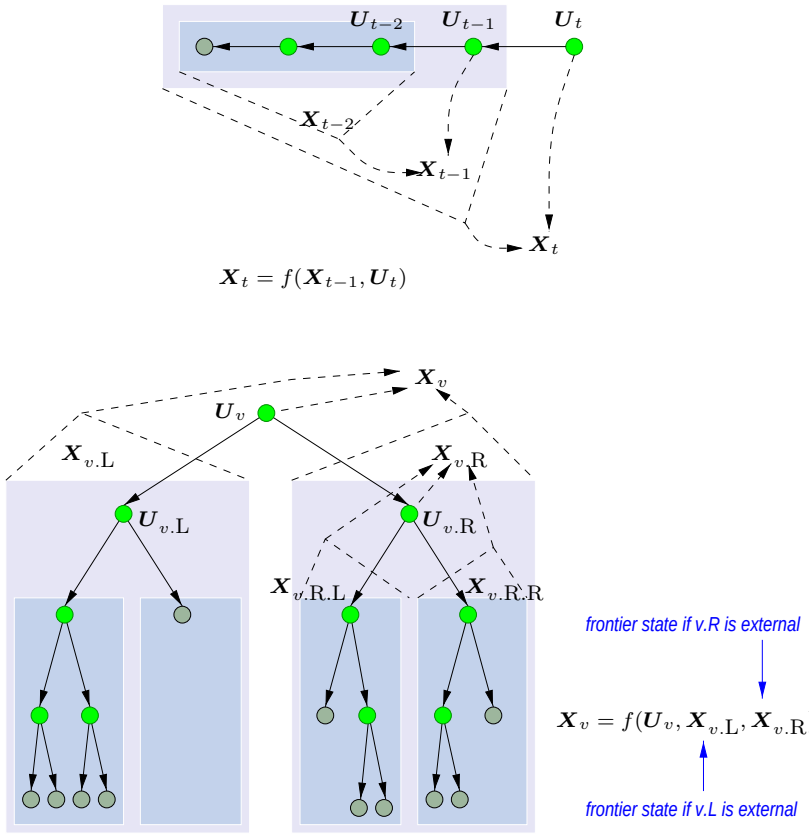


Fig. 8. Recursive state update scheme. Top: temporal domains. Bottom: in a binary-tree domain.

is known. An example for the simple case where # is the class of binary trees is shown on the bottom of Fig. 8). Note that state updating proceeds in a bottom-up way, according to any reverse topological sort of vertices in $\boldsymbol{U}$. Algorithms 1,2 illustrate state updating assuming that the input graph is a DPAG with a supersource. Note that, since in general topological sort is not unique, some states may be actually updated in parallel, as shown in Fig. 9.

The last important concept about recursive transductions is *stationarity*. Again, this concept is often encountered in systems theory for sequences, and can be generalized to graphical dynamical systems. In particular, a recursive transduction is said to be stationary if $f(\boldsymbol{X}_{\mathrm{pa}[v]}, U_v, v) = f(\boldsymbol{X}_{\mathrm{pa}[v]}, U_v, w)$ and $g(X_v, U_v, v) = g(X_v, U_v, w)$ for each pair of vertices v, w.

Algorithm 1 UPDATE($\boldsymbol{U}, \boldsymbol{X}, v$)

```
if v is an external vertex then
    X_v ← X_0 (the frontier state);
else
    for k ← 1, ..., m do
    w ← k-th child of v;
    if w is not marked as "reached" then
        UPDATE(U, X, w);
    Mark v as "reached";
    X_v ← f(X_ch[v], U_v, v)
```

Algorithm 2 RECURSIVE-STATE-UPDATE($\boldsymbol{U}, \boldsymbol{X}$)

```
s ← supersource(U);
Mark all vertices of U as "not reached";
UPDATE(U, X, s);
```

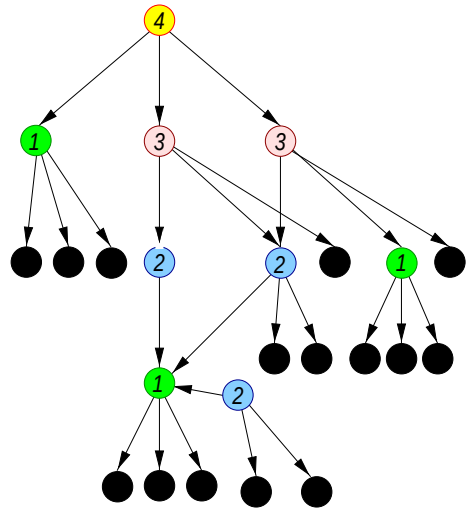


Fig. 9. Example of scheduling for the recursive state update. States labeled by the same number can be updated in parallel.

4.3 Supersource Transductions

Quite often, the learning problem we are faced with simply consists of predicting a scalar quantity as a function of an input data structure. For example, this is the case when we want to classify a data structure, or when we want to associate a continuous score with a data structure (like, e.g., the heuristic evaluation function for automated reasoning described in Section 2.2). In these cases, the transduction is defined as $\tau : \mathcal{U}^{\#} \to \mathcal{Y}$, i.e., we may think that the skeleton of

the output structure is the trivial graph having a single vertex and no edges. The recursive state representation described above can be still conveniently used in this case. In particular, we shall assume that a state variable exists, that is updated through the data structure according to equation 3. The best way to understand the bottom line of the method is probably to think about accepting automata for strings. Such automata update their state by making transitions, controlled by symbols sequentially read from the input string. Then, a binary output is emitted at the end of the string, depending whether the final state belongs or not to the set of accepting states. In order to generalize this approach to graphs, we need to locate a "final" vertex s in the graph. The obvious requirement is that the state in s summarize all the information contained in the input data structure $\boldsymbol{U}$. Therefore, s must correspond to the supersource of $\boldsymbol{U}$. A supersource transduction is then a function $\tau() : \boldsymbol{U} \in \mathcal{U}^{\#} \rightarrow Y \in \mathcal{Y}$ defined as

$$X_v = f(\boldsymbol{X}_{\mathrm{ch}[v]}, U_v, v) \tag{5}$$

$$Y = g(X_s). \tag{6}$$

Note that if $\boldsymbol{U}$ does not possess a supersource, it is always possible to conventionally add a supersource to $\boldsymbol{U}$, in a way that minimizes the number of new edges [7].

5 Graphical Models for Recursive Transductions

We now develop a graphical formalism for describing IO-isomorph transductions that admit a recursive state space representation. The main advantages of this formalism is that transducers can be easily interpreted as networks, thus simplifying, at least at a conceptual level, the description of specific architectures and learning algorithms.

5.1 Recursive Networks

Dynamical systems in engineering are often described by means of interconnected block diagrams or networks. For example, sequential circuits in digital electronics are represented as networks whose more important components are logic gates and delay elements, implemented by means of flip-flops. Similarly, filters in digital signal processing are schematically represented as networks whose main components are adders, multipliers, and delay elements. Neural networks can be sometimes regarded as circuits and, indeed, recurrent neural networks are often depicted using delay elements along cyclic connections. We present here a similar graphical formalism for representing transductions that operate on structured domains.

The more important component that has to be generalized is the unitary time delay element. From a mathematical point of view, a delay element is modelled

by means of the shift operator q^{-1}, which is regarded as an algebraic operator[4] that can be applied to a temporal variable X_t and is defined as $q^{-1}X_t = X_{t-1}$. The power of the shift operator is defined inductively as $q^0 = 1$, $q^{-k} = q^{-1}q^{k-1}$ and is used to represent multiple delays. The inverse shift operator (or advance element in a circuit) q can also be defined and it implies non causal dependencies.

A generalized shift operator can be introduced for m-DPAGs as follows. Let $\boldsymbol{X} \in \mathcal{X}^{\#}$. Then, for $k = 1, \ldots, m$ $q_k^{-1}X_v$ is the label attached to the k-th child of vertex v, if such a child is an internal node, or $q_k^{-1}X_v = \emptyset$ if the k-th child of v belongs to the frontier. An intuition to acquire a better understanding of the generalized shift operator for graphs is to remember that sequences are linear lists, and to think about the usual computer implementation based on pointers. Then the operation $q^{-1}X_t$ may be thought of as follows: given the variable X_t, take the vertex t labeled by such variable, follow the pointer to the next element (thus moving to vertex $t-1$) and return the label there found. The operations performed on m-DPAGs are similar, but now instead of just one child there are m children. It should be noted that composition of generalized shift operators is not commutative, as shown in Fig. 10.

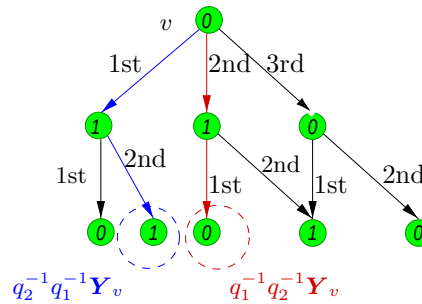


Fig. 10. The composition of generalized shift operators is not commutative. In this example $q_2^{-1}q_1^{-1}Y_v$ yields the label 1, while $q_1^{-1}q_2^{-1}Y_v$ yields the label 0.

Now we can define the recursive network associated with a transduction $\tau()$. We shall assume the following hypotheses:

1. $\tau()$ admits a recursive state space representation;
2. input, state, and output are uniformly labeled.

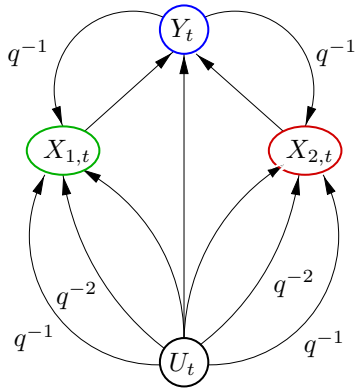
The recursive network of $\tau()$ is a directed graph where:

- vertices are marked with representative variables;
- edges are marked with generalized shift operators;

[4] Other discrete time operators have been introduced in the temporal domain. For example, the gamma operator [27] is defined by $\gamma = \frac{q-1+\mu}{\mu}$ being μ a constant between 0 and 1. A good review of discrete time operators can be found in [28].

– an edge (A_v, B_v), with label q_k^{-p}, means that for each v in the vertex set of the input structure B_v is a function of $q_k^{-p} A_v$.

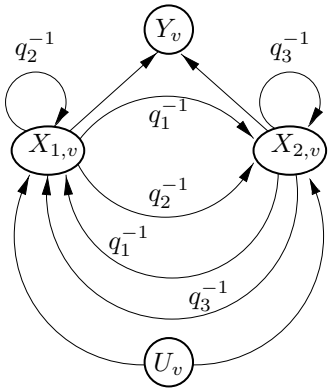
Figure 11 shows two examples of recursive networks. The first example refers to temporal domains and the network shown on the top of Fig. 11 corresponds to the transduction operated by a NARX neural network [29]. The other example is a transduction that operates on m-DOAGs.



$$Y_t = g(X_{1,t}, X_{2,t}, U_t)$$

$$X_{1,t} = f(U_t, q^{-1}U_t, q^{-2}U_t, q^{-1}Y_t)$$

$$X_{2,t} = f(U_t, q^{-1}U_t, q^{-2}U_t, q^{-1}Y_t)$$



$$Y_v = g(X_{1,v}, X_{2,v})$$

$$X_{1,v} = f(U_v, q_1^{-1}X_{2,v}, q_2^{-1}X_{1,v}, q_3^{-1}X_{2,v})$$

$$X_{2,v} = f(U_v, q_1^{-1}X_{1,v}, q_2^{-1}X_{1,v}, q_3^{-1}X_{2,v})$$

Fig. 11. Two examples of recursive networks.

5.2 Encoding Networks

As described above, the variables involved during the processing of a given data structure $\boldsymbol{U}$ can be locally represented using the recursive network associated with the transduction. If we desire a global representation of the whole set of

involved variables, we may use the recursive network as a template which is unfolded (i.e., expanded) according to the skeleton of the input data structure (that, by IO-isomorphism assumption, matches the skeleton of the output structure). The resulting labeled graph (function of both the recursive network $N(\mathcal{T}, \#)$ and the input-output data structures $\boldsymbol{U} \in \mathcal{U}^{\#}$,$\boldsymbol{Y} \in \mathcal{Y}^{\#}$) is called *encoding network* of the transduction, denoted $\mathcal{E}(N, \boldsymbol{U}, \boldsymbol{Y})$. Nodes and edges of $\mathcal{E}(N, \boldsymbol{U}, \boldsymbol{Y})$ are constructed as follows:

- The vertex set of $\mathcal{E}$ is the Cartesian product of the vertex sets of N and skel($\boldsymbol{U}$). For each $v \in$ vert(skel($\boldsymbol{U}$)), $U_{i,v}$ denotes the i-th input variable at node v, $X_{i,v}$ denotes the i-th state variable at node v, and $Y_{i,v}$ denotes the i-th output variable at node v.
- The edge set of $\mathcal{E}$ is obtained as follows. Let $A_{i,v}$ $B_{j,w}$ denote two vertices of $\mathcal{E}$. The directed edge $(A_{i,v}, B_{j,w})$ is present in $\mathcal{E}$ if and only if the edge (A_i, B_j) is present in N and is labeled by an expression $L(\boldsymbol{q}^{-1})$ such that $L(\boldsymbol{q}^{-1})\boldsymbol{U}_w = \boldsymbol{U}_v$.

Simple examples of the unrolling operation are shown in Fig. 12, for the cases of sequences and binary trees. It can be noticed that unrolling is simply a compact graphical description of the recursive state update algorithm that we have discussed in Sec. 4.2. In a certain sense, the encoding network can be seen as a pre-compiled version of Alg. 2.

Algorithm 3 builds the skeleton of the encoding network associated with the recursive network N and the skeleton $S = \text{skel}(\boldsymbol{U}) = \text{skel}(\boldsymbol{Y})$ of the input-output DOAGs. A more complex example of encoding network construction is shown in Fig. 13. Note that arrows in the encoding networks are reversed with respect to the direction of arrows in the input data structure. This is because computation in recursive causal transductions proceeds from the frontier to the supersource. Shift operators can be also applied to nodes of the encoding network. In particular, $q_k^{-1}\boldsymbol{X}_v$ returns the state label attached to the k-th child of v in the skeleton of $\boldsymbol{U}$ (which is the k-th parent of $\boldsymbol{X}_v$ in the encoding network).

6 Models for Learning Recursive Transductions

6.1 Models Based on Neural Networks

Let us consider a recursive transduction operating on $\mathcal{U}^{\#^m}$, described by equations 3. In a connectionist realization of the transduction it is assumed that the state X_v is a continuous vector (i.e., $\mathcal{X} \equiv \mathbb{R}^n$). Moreover, the state transition function f and the output function g are realized by *static* neural networks, such as multilayered perceptrons or networks with radial basis function units. The network for realizing f has $n \cdot m \cdot |\mathcal{U}|$ input units and n output units. The network for realizing g has $n \cdot |\mathcal{U}|$ input units and $|\mathcal{Y}|$ output units. When f and g are realized by neural networks, they are parameterized by a set of connection weights that can be tuned to adapt the behaviour of the model according to the training data.

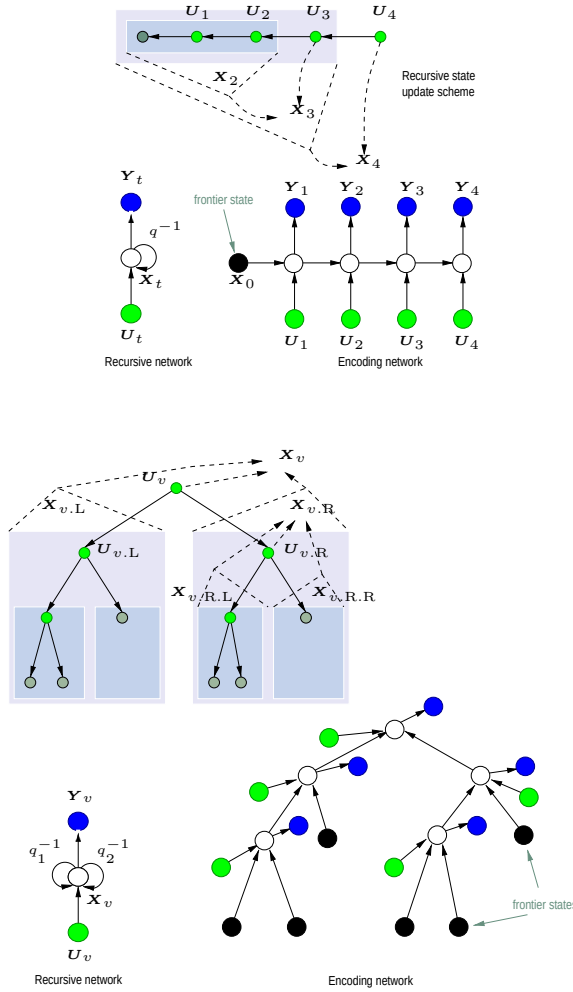


Fig. 12. Encoding networks for sequential (top) and binary-tree (bottom) domains.

It should be noted that when we build the encoding network for a transduction realized by neural networks, we obtain a feedforward neural network whose actual topology is determined by the input structure. If stationarity is assumed (and this is the typical case), the same set of weights is replicated for each vertex of the input structure. This may be actually thought of as a form of weight sharing for the encoding network.

Supervised learning is accomplished as usual, by defining an error function that takes into account the difference between actual and desired output, for each vertex v. Looking at the encoding network obtained in this way makes the problem of training quite straightforward. Indeed, using standard backpropaga-

Algorithm 3 BUILD-ENCODING-NETWORK(N, S)

```
1  vert(E)←∅
2  egd(E)←∅
3  foreach v ∈ vert(S) do
4      foreach A_i ∈ vert(N) do
5          vert(E)←vert(E) ∪ {A_{i,v}}
6  foreach (A_i, A_j, L(q^{-1})) ∈ egd(N) do
7      foreach v ∈ vert(S) do
8          if L(q^{-1})v ≠ nil then
9              egd(E)←egd(E) ∪ {(L(q^{-1})A_{i,v}, A_{j,v})}
10         else
11             egd(E)←egd(E) ∪ {(X_i^F, A_{j,v})}
12 return E
```

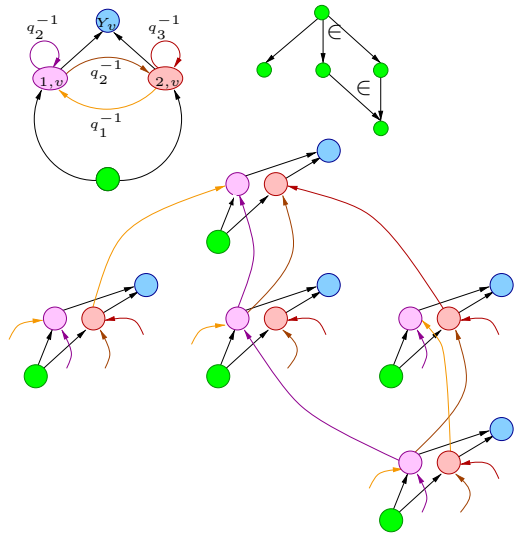


Fig. 13. A relatively complex encoding network for a 3-DPAG. Left, center, and right edges are associated to child 1, child 2, and child 3, respectively.

tion on the encoding network (with constraints originated by the sharing due to stationarity) yields the correct gradients of the error function with respect to the weights [30]. As a special case, this procedure applied to sequences corresponds to the well known backpropagation through time algorithm [31] commonly used for training recurrent neural networks.

6.2 Probabilistic Graphical Models

Models for probabilistic transductions can be realized in different ways. The simplest idea is to give a probabilistic interpretation to a recursive neural network

model, in the same spirit as it is commonly done with feedforward neural networks [32]. A more radical approach consists of building a model with stochastic graphical dynamics. Essentially, the idea is to generalize HMMs and input-output HMMs to graphs, exactly like it has been done with recurrent neural networks. The recursive state space representation described by equations 3 and 4 can be modified in the following stochastic version:

$$P(\boldsymbol{U}, \boldsymbol{Y}) = P(\boldsymbol{U}) \prod_v P(Y_v|X_v, U_v) P(X_v|\boldsymbol{X}_{\mathrm{ch}[v]}, U_v) \tag{7}$$

where $P(X_v|\boldsymbol{X}_{\mathrm{ch}[v]}, U_v)$ are the *state transition densities* and $P(Y_v|X_v, U_v)$ are the *emission densities*. For those readers who are familiar with probabilistic graphical models, it is immediate to recognize that equation 7 is the factorization formula for a Bayesian network [33, 34] modelling the joint densities over the input output labels involved in the transduction. In a generic Bayesian network that models the joint density for a set of variables $\{A_1, \ldots, A_n\}$, the factorization formula is in fact

$$P(A_1, \ldots, A_n) = \prod_i P(A_i|\mathrm{Pa}[A_i]). \tag{8}$$

Therefore, it is easy to see that the Bayesian network associated with the factorization 7 is isomorph to the encoding network for the transduction, as we have defined in Sec. 5.2. In the special case of sequences, equation 7 reduces to the joint density for input-output HMMs. In fact, it is now widely recognized that Markovian models are just a special case of probabilistic belief networks [35].

Inference and learning algorithms might be inherited from general algorithms for inference and learning in Bayesian networks. However, densely connected data structures yield densely connected Bayesian networks, for which inference is known to be intractable [36]. Methods for circumventing this problem might take advantage of approximated inference, as done for example in [37], and are currently under investigation.

References

1. L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.
2. Y. Bengio and P. Frasconi, "Input-output HMM's for sequence processing," *IEEE Transactions on Neural Networks*, vol. 7, pp. 1231–1249, September 1996.
3. J. Thacher, "Tree automata: An informal survey," in *Currents in the Theory of Computing* (A. Aho, ed.), pp. 143–172, Englewood Cliffs: Prentice-Hall Inc., 1973.
4. J. B. Pollack, "Recursive distributed representations," *Artificial Intelligence*, vol. 46, no. 1-2, pp. 77–106, 1990.
5. A. Sperduti, "Labeling RAAM," *Connection Science*, vol. 6, no. 4, pp. 429–459, 1994.
6. A. Sperduti, "Encoding labeled graphs by labeling RAAM," in *Advances in Neural Information Processing Systems* (J. D. Cowan, G. Tesauro, and J. Alspector, eds.), vol. 6, pp. 1125–1132, Morgan Kaufmann Publishers, Inc., 1994.

7. A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, 1997.
8. A. Sperduti, "Neural networks for processing data structures," 1997. (in this volume).
9. P. Frasconi, M. Gori, and A. Sperduti, "A general framework for adaptive processing of data structures," Tech. Rep. DSI-RT15/97, Università di Firenze, 1997. (submitted).
10. K. S. Fu, *Syntactic Pattern Recognition and Applications*. Englewood Cliffs, N.J: Prentice-Hall, 1982.
11. T. Pavlidis, *Structural Pattern Recognition*. Springer-Verlag, 1977.
12. R. C. Gonzalez and M. G. Thomason, *Syntactical Pattern Recognition*. Addison-Wesley, 1978.
13. R. J. Schalhoff, *Pattern Recognition: Statistical, Structural and Neural Approaches*. John Wiley & Sons, 1992.
14. E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, pp. 302–320, 1978.
15. D. Angluin and C. H. Smith, "A survey of inductive inference: Theory and methods," *ACM Comput. Surv.*, vol. 15, pp. 237–269, Sept. 1983.
16. C. Omlin and C. Giles, "Constructing deterministic finite-state automata in recurrent neural networks," *Journal of the ACM*, vol. 43, no. 6, pp. 937–972, 1996.
17. M. Casey, "The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction," *Neural Computation*, vol. 8, no. 6, pp. 1135–1178, 1996.
18. S. C. Kremer, *A Theory of Grammatical Induction in the Connectionist Paradigm*. PhD thesis, Dept of Computer Science, University of Alberta, Canada, 1996.
19. N. Nilsson, *Principles of Artificial Intelligence*. Palo Alto: Tioga, 1980.
20. R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.
21. L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: Introduction and Applications*. Prentice-Hall, 1984.
22. C. Goller, *A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems*. PhD thesis, Technical University Munich, Computer Science, 1997.
23. J. Oncina, P. Garcia, and E. Vidal, "Learning subsequential transducers for pattern recognition interpretation tasks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, pp. 448–458, May 1993.
24. R. . Neco and M. L. Forcada, "Beyond mealy machines: learning translators with recurrent neural," in *Proc. WCNN'96 (World Congress on Neural Networks)*, (San Diego, CA), pp. 408–411, 1996.
25. Y. Bengio and F. Gingras, "Recurrent neural networks for missing or asynchronous data," in *Advances in Neural Information Processing Systems* (M. Mozer, D. Touretzky, and M. Perrone, eds.), vol. 8, The MIT Press, 1996.
26. M. L. Forcada and R. C. Carrasco, "Learning the initial state of a second-order recurrent neural network during regular-language inference," *Neural Computation*, vol. 7, no. 5, pp. 923–930, 1995.
27. B. de Vries and J. C. Principe, "The gamma model – A new neural net model for temporal processing," *Neural Networks*, vol. 5, pp. 565–576, 1992.
28. A. D. Back and A. C. Tsoi, "A comparison of discrete-time operator models and for nonlinear system identification," in *Advances in Neural Information Processing Systems* (G. Tesauro, D. Touretzky, and T. Leen, eds.), vol. 7, pp. 883–890, The MIT Press, 1995.

29. H. Siegelmann, B. Horne, and C. Giles, "Computational capabilities of recurrent narx neural networks," *IEEE Trans. on Systems, Man and Cybernetics*, 1997. In press.
30. C. Goller and A. Küchler, "Learning task-dependent distributed structure-representations by backpropagation through structure," in *IEEE International Conference on Neural Networks*, pp. 347–352, 1996.
31. R. J. Williams and D. Zipser, "Gradient-based learning algprithms for recurrent networks and their computational complexity," in *Backpropagation: Theory, Architectures, and Applications* (Y. Chauvin and D. E. Rumelhart, eds.), Lawrence Erlbaum Associates, 1995.
32. D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin, "Backpropagation: The basic theory," in *Backpropagation: Theory, Architectures and Applications*, pp. 1–34, Hillsdale, NJ: Lawrence Erlbaum Associates, 1995.
33. J. Pearl, *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. Morgan Kaufmann, 1988.
34. S. Lauritzen, *Graphical Models*. No. 17 in Statistical Science Series, Oxford Clarendon, 1996.
35. P. Smyth, D. Heckerman, and M. I. Jordan, "Probabilistic independence networks for hidden markov probability models," *Neural Computation*, vol. 9, no. 2, pp. 227–269, 1997.
36. G. F. Cooper, "The computational complexity of probabilistic inference using Bayesian belief networks," *Artificial Intelligence*, vol. 42, pp. 393–405, 1990.
37. Z. Ghahramani and M. Jordan, "Factorial hidden Markov models," *Machine learning*, 1997. (to appear).