Chapter 19

From Sequences to Data Structures: Theory and Applications

Paolo Frasconi, Marco Gori, Andreas Küchler and Alessandro Sperduti

19.1 Introduction

This whole book is devoted to architectures for learning and processing sequential data. Often, these architectures extend feedforward neural networks from fixed-size vectors to sequences. In this chapter, we further extend the *type* of data a neural network can deal with, from sequences to labeled graphs.

Developing connectionist networks capable of dealing with graphs (rather than fixed-size vectors or sequences) is interesting because there are several domains in which data are (or can be) organized into a structured form. This means that part of the information which is relevant for solving a given problem is encoded, sometimes implicitly, into the relationships between basic entities. This holds for problems from chemistry, software engineering, pattern recognition, natural language processing, and problem solving in artificial intelligence.

When faced with this kind of data, it might be tempting to convert graphs into sequences (an operation that can always be performed with no loss of information) and invoke standard sequence learning methods, rather than graph learning methods. However, this is not always appropriate. The conversion of graphs into sequences breaks explicit relationships among basic entities and turn them into implicit (often long-ranged) dependencies between sequence elements. This is very likely to generate a hard learning task, especially if the resulting sequences have long-ranged dependencies (see Chapter 14). Moreover, the conversion can increase the requirements of the model in terms of expressive power.

Another reason for the interest in these models is related to the development of hybrid connectionist-symbolic models. A very popular view of hybrid systems is one in which numerical data are processed by a sub-symbolic module, while structured data are processed by the symbolic counterpart of the system. Unfortunately, because of the different nature of numerical and structured representations, a tight integration of the different components seems to be very difficult. In some approaches to hybrid systems, the role of the prior knowledge is that of providing a partial specification of the transduction to be learned (see e.g. Chapter 10). Learning data structures offers a novel view of hybrid systems, in which the incorporation of symbolic knowledge involves the nature of the data themselves.

This chapter is organized as follows. Section 19.2 briefly summarizes the history of connectionist structure processing. Section 19.3 reviews the basic formalism for extending recurrent neural networks (for sequences) to *recursive* neural networks (for trees and directed acyclic graphs). Finally, section 19.4 surveys most relevant applications recently carried out in the field and provides some general guidelines for the application of the described approach.

19.2 Historical Remarks

In this section, we survey the longstanding discussion on connectionist structure processing (e.g. distributed vs. local representation, tensor products, holographic reduced representation, (L)RAAM), and illustrate the deficiencies of earlier approaches.

One of the criticisms to connectionism, mainly elaborated by Fodor and Pylyshyn [Fodor and Pylyshyn, 1988], is that it lacks the flexibility and representational adequacy needed for higher level cognitive tasks. Several researchers have tried to reply by focusing their efforts to demonstrate how symbolic structures like lists, trees, and stacks can be represented and manipulated in a connectionist system. The goal of these researchers is to provide evidence of the potential of the connectionist approach to handle domains of structured tasks. The common background of their ideas is the search for a realization of the distal access ability and consequently of the compositionality one. BoltzCONS [Touretzky, 1990] is an example of how a connectionist system (i.e. Boltzman machine) can handle symbolic structures. It is based on parallel associative retrieval and it differs from other connectionist systems because it constructs and modifies composite symbol structures dynamically, by representing them as activity patterns rather than as weights. It uses distributed representations of linked lists, loaded in coarse-coded memories * [Rosenfeld and Touretzky, 1987],

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

as basic representational elements. LISP's *car*, *cdr*, and *cons* functions are used as basic operations and links are implemented by associations. The associative retrieval capabilities of BolzCONS support several computational primitives. Examples are the instantaneous access to an element of a list, the capacity to rapidly access parts of a symbol structure based on closest match (rather than symbolic exact match), and efficient retrieval using multiple cues (in general, supplying more constraints causes a connectionist model to settle faster; instead, a conventional machine using hash table representations will be slowed down). The full architecture supports direct representations of arbitrary tree structures, and it can perform complex pointer manipulations using multiple buffers operating simultaneously on its memory. It must be noted that the dynamic manipulation of structures by BoltzCONS requires the same allocation and reclamation problems usually met in conventional symbolic systems. Moreover, even if coarse-coding representations gives successful results, there remain some problems associated with this representation strategy. Firstly, coarse-coding requires expensive and complex access mechanisms (i.e. pullout networks). Secondly, coarse-coded memories can only simultaneously instantiate a small number of representational elements before spurious ones are introduced. Finally, the coarse-coded memory of BoltzCONS needs a huge number of units because it utilizes binary codes.

The above problems, together with the observation that BoltzCONS needs a large amount of human effort to design, to compress and to tune the representations, have stimulated Pollack [Pollack, 1990b] to design the Recursive Auto-Associative Memory (RAAM) architecture. The RAAM system uses back-propagation (on an encoder feed-forward network; see [Rumelhart et al., 1986] Vol. 1, p. 335) to discover compact recursive distributed representations for fixed values trees. These representations are very interesting because they synthesize the characteristics of an item (categorical features) preserving its individual peculiarity (distinctive features). Moreover, they utilize real values over few units and they can be composed selectively by a potentially very large number of primitive elements. Other advantages are related to the fact that they are developed mechanically, their aggregation mode is compositional, and their access mechanisms are simple and deterministic. However the most relevant aspect of this work is that RAAM can devise representations of trees as numeric vectors. If inference over trees might be performed by numerical transformation (i.e. neural networks) over their numerical representation, very fast and cheap inference engines would be built (see [Chalmers, 1990]). Sperduti proposes an evolution of the RAAM network able to encode labeled graphs, i.e., the LRAAM model [Sperduti, 1994]. With this model it is possible to encode labeled graphs, even containing cycles. However, cyclic graphs that are not in the training set are difficult to be evaluated because it is not clear which initial representation for the constituents of a cycle should be used.

In the context of natural language processing, the appropriateness of LRAAM representations for subsequent classification tasks has been shown by Cadoret [Cadoret, 1994], where the distributed representations of syntactical trees devised by an LRAAM are automatically classified according to the typology of dialogue acts. LRAAMs have been used also by Ellingsen [Ellingsen, 1997] to build up a connectionist-based system for analogical mapping of object-oriented specifications. Recently, Callan and Palmer-Brown [Callan and Palmer-Brown, 1997] have proposed a fast incremental training procedure to train linear RAAMs.

A more formal characterization of representations of structures in connectionist systems has been developed by Smolensky [Smolensky, 1990]. He reduces the problem of representing structured objects to three subproblems: decomposing the structures via roles, representing conjunctions, and representing variable/value bindings. The representation of variable/value bindings is obtained through tensor algebra. In the tensor product representation, both the variables and the values can be arbitrarily distributed[†], enabling representations in which every unit is part of the representation of every constituent in the structure. This generality allow one to express existing cases of connectionist representations as particular cases of the tensor representations. Smolensky discusses several features of the tensor product representation. These include the graceful saturation and exact retrieval for non-saturated memories, the representation of continuous structures as well as of finite ones, the representation of symbolic operators and recursive structures, the possibility to define and analyze optimal role vectors, and the possibility to use values bounded to one variable as a variable.

Hinton [Hinton, 1990] observes that reduced representations of structured objects in connectionist systems are related to the problem of mapping part-whole hierarchies into connectionist networks. The scheme he proposes considers two quite different methods for performing inference. Simple "intuitive" inferences can be performed by a single settling of a network without changing the way in which the world is mapped into the network. More complex "rational" inferences involve a sequence of such settlings with mapping changes after each settling. He discusses three approaches to map a part-whole hierarchy into a finite amount of parallel hardware: fixed mappings, within-level timesharing, and between-level timesharing. In the first approach, each object in the hierarchy is always mapped into a pattern of activity in the same set of units, and each set of units is always used to represent the same object (one-to-one mapping). In the second one, many different objects at the same level can be mapped into the same set of units in the serial recognition apparatus, but whenever one of these objects is recognized, it is represented in the same units. Finally, between-level timesharing allows many different objects at the same level to be mapped into different sets of units depending on the level at which attention is focused. The idea that he proposes is to use role-specific reduced representations to implement pointers. This kind of pointer, as Pollack's reduced distributed representations, contains relevant information about the object they point to. In this

[†]The choice to have distributed representation of variables (i.e. roles) is controversial. Hinton [Hinton, 1990] believes that in a nonlinear system it is probably easier to make use of the information about the fillers of roles if this information is localized.

way, some type of computations over objects can be performed directly on the reduced representation, without accessing the full description of the object. In this framework, a general computation results from the combination of slow sequential access to full descriptions and fast parallel constraint-satisfaction using reduced descriptions ("intuitive" inference). If the appropriate representations are known, an "intuitive" inference allows making a lot of useful computation. When the computational task is difficult, a sequence of settlings is executed, and after each settling, the mapping between the world and the network is changed. The possibility to change the mapping allows the application of the knowledge of the system to any part of the task. A very interesting way of implementing reduced representations is by using a circular convolution together with a clean-up memory (Holographic Reduced Representations [Plate, 1995]).

Another example of neural system able to encode small labeled graphs representing relationships among entities is the DUAL system proposed by Dyer [Dyer, 1991]. The idea is to have two networks, one responsible for the encoding of the relationships between one particular entity and the others, and one which devises a compressed representation of the weights of the first network. The compressed representation is employed as follows. In the first network, it serves as target for the training. In the second network, when information about the relationships of an entity with the others is needed, it serves as a pattern of activation for the hidden units to obtain the weights for the first network.

All the above systems, with few exceptions, share the idea of encoding a structure into a *flat* vectorial representation. The main problem in doing that is the increase in size of the representation accordingly with the size of the structure. Only the RAAM and HRRs adopt a compression strategy which allows the representations to stay the same predefined size, regardless of the encoded structure size. Of course, the ability to maintain constant size representations for structures of different size is paid in terms either of the maximum number of structures which can be safely encoded into the system or with loss of information about specific aspects of the structures (basically labels attached to the structure's nodes and topology).

19.3 Adaptive Processing of Structured Information

In this section we review the general theory of recursive neural networks (more details can be found in [Frasconi et al., 1998]). We begin by giving a definition of the learning domain. Then we introduce the general computational model and analyze some of its properties. Finally we discuss learning algorithms. The reader should bear in mind that we are extending recurrent networks. When applied to linear lists, the theory presented in this section smoothly reduces to the theory of recurrent networks.

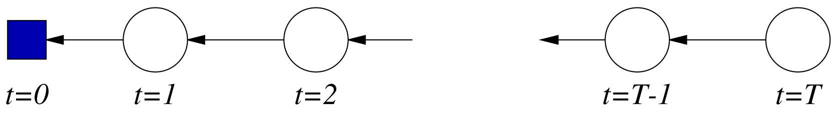

Figure 19.1: A sequence seen as a data structure built upon 1-SDOAGs. The black square denotes the missing child of node 1.

19.3.1 Structured domains

The domains we consider are described by labeled graphs. In the following, we give a formal definition of these domains. Let us consider a universe $\boldsymbol{U}$ described by a (possibly infinite) set of categorical and numerical variables and let us assume that n equivalence relations are defined among them. Variables within the same equivalence class are supposed to have the same type and semantics. The set of representatives for these equivalence classes is called *label space*, denoted $\mathcal{U}$. A label *on* $\mathcal{U}$ is a tuple of n variables in the universe, each belonging to exactly one of the equivalence classes specified by $\mathcal{U}$. A *data structure* over the label space $\mathcal{U}$ is a graph with labels attached to its nodes. An edge (u, v) in the graph indicates that variables in the labels attached to u and v are related.

A *skeleton class* is a set of unlabeled graphs that satisfies some specified topological conditions. For example, the class "DAG" is formed by directed acyclic graphs. A generic class of skeletons is denoted by the symbol $\#$. Given a label space, $\mathcal{U}$, and a class of skeletons, $\#$, we can introduce a domain of data structures composed by all the graphs with topology in $\#$ and labels on $\mathcal{U}$. Such domain is denoted $\mathcal{U}^{\#}$.

Some skeleton classes have particular importance in the following discussion. The class DOAG is formed by directed ordered acyclic graphs, where "ordered" means that a total order is defined for the edges leaving any given node. In the case of classification tasks, graphs are required to possess a *supersource*, i.e. a parentless node s such that any other node can be reached from s. This requirement is due to the specific computational model, which is introduced in Section 19.3.2. We call SDOAG the class of DOAGs with a supersource. By definition, SDOAGs are connected. The class k-DOAG is a subclass of DOAG whose graphs have maximum out-degree k. The class DPAG is formed by directed positional acyclic graphs with a supersource. Here "positional" means that a bijection between integers and edges leaving any given node is defined (hence, each child has its own position within the set of children of a node). In the case of DPAGs, the special *nil* symbol is used to denote missing children.

To clarify these definitions, let us consider some examples. First, we note that data structures are a generalization of the sequence data type studied so far in this book. Indeed, if $\#$ is the class of connected 1-DOAGs, then graphs have a linear chain topology and each node can be associated with a particular discrete time point. In particular, *finite* sequences are built upon 1-SDOAGs in

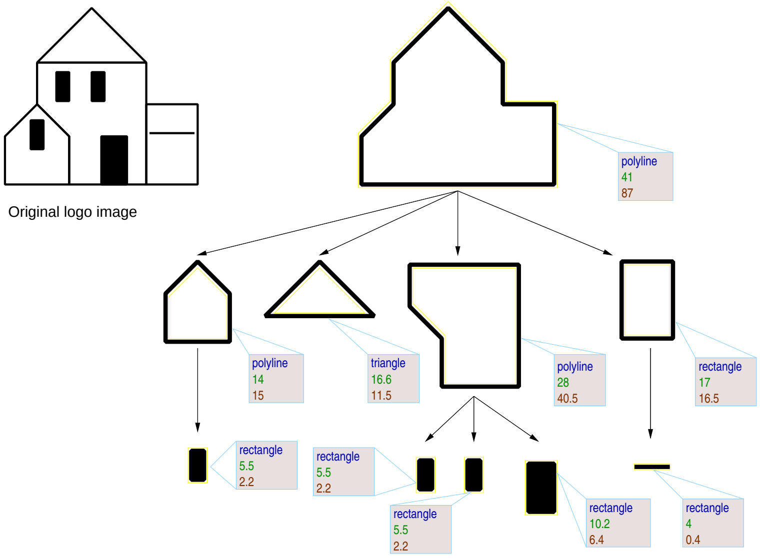

Figure 19.2: A drawing (left) represented by a tree with local features forming node labels.

which the supersource corresponds to the last time index of the sequence. In this case, the label is a tuple of variables which are observed at a particular discrete time point. Strings over a finite alphabet Σ (as used, for example in Chapters 6, 7, 8, 9, 10, and 12) are a special case of data structures having skeleton in the class of connected 1-DOAG (see Figure 19.1) and the label space $\mathcal{Y}$ is a single class of variables with realizations in Σ. Interestingly, the notation $\mathcal{Y}^{\#}$ can be seen as a generalization for the notation commonly used for the Kleene closure of Σ, i.e. Σ^*.

The second example we consider is the contour-tree representation of images [Cortelazzo et al., 1994]. In this case, the topology of data structures is the class of rooted ordered trees, obtained by recursively decomposing the image into sub-constituents, as shown in Figure 19.2. The label space in this example contains three types of variables, named `shape`, `perimeter` and `area` respectively. The first variable is categorical, with realizations in the set $\{\texttt{polyline}, \texttt{triangle}, \texttt{rectangle}\}$. The other two variables are numerical. Each node of the tree is labeled by a triplet of variables instantiated with the shape, the perimeter, and the area of the sub-constituent.

19.3.2 The computational model

Here we give a description of the computational models suitable for the processing of structured information. Our discussion is focused on the supervised learning task, formalized as the problem of estimating the conditional probability $P(\boldsymbol{Y}|\boldsymbol{U})$, where $\boldsymbol{Y}$ and $\boldsymbol{U}$ are generic objects. For example, in the classification task (or concept learning task), $\boldsymbol{Y}$ is a variable representing one of n categories and $\boldsymbol{U}$ is an object in the instance space. As seen in Chapter 2, feedforward neural networks can be employed for estimating the conditional probability $P(\boldsymbol{Y}|\boldsymbol{U})$ if $\boldsymbol{U}$ is a tuple of attributes (see also [Bishop, 1995]). Within the framework proposed in Section 19.3.1, a tuple of attributes can be seen as a trivial data structure built upon the skeleton class formed by graphs with a single node. Throughout this book, we have discussed how neural networks can be employed for processing sequences. The basic recurrent neural network architectures are based on the following recursive state space representation:

$$\begin{aligned} \boldsymbol{a}(t) &= f(\boldsymbol{a}(t-1), \boldsymbol{I}(t)) \\ \boldsymbol{y}(t) &= g(\boldsymbol{a}(t)) \end{aligned} \tag{19.1}$$

where $\boldsymbol{a}(t)$ is the state vector formed by collecting hidden dynamic units' activations, $\boldsymbol{I}(t)$ is the input vector, and $\boldsymbol{y}(t)$ is the output prediction at discrete time t. The state transition function f and the output function g are implemented by feedforward neural networks. As a variant, the output can be computed adding an explicit dependency on the input, i.e. $\boldsymbol{y}(t) = g(\boldsymbol{a}(t), \boldsymbol{I}(t))$ (this is similar to the distinction between Mealy and Moore models in the case of finite state machines). Within our framework, sequences are data structures built upon the skeleton class formed by linear-chain graphs (in this case, nodes can be associated with discrete time steps). The computation described by Equations 19.1 can be explained as follows. The state associated with a given node t is a function of the state found in the child (i.e., $t-1$) and the current input (label attached to node t). The output associated with a given node t is a function of the state associated with t and the label attached to t. Since we are assuming the use of neural networks, labels are represented by means of real vectors $\boldsymbol{I}(t)$ (for example, collecting numerical attributes as they are and encoding categorical attributes according to some convention such as one-hot).

The whole processing performed by a dynamic network on sequences can be conveniently described by the graphical model shown in Figure 19.3. Note that the base step of recursion 19.1 (the initial state $\overline{\boldsymbol{a}}$) corresponds to the only missing child in the linear chain associated with the sequence.

A very similar recursive representation can be introduced for more general skeleton classes. In particular, let us focus on k-DOAGs and consider the following computational scheme:

$$\begin{aligned} \boldsymbol{a}(v) &= f(\boldsymbol{a}(\mathrm{ch}[v]), \boldsymbol{I}(v)) \\ \boldsymbol{y}(v) &= g(\boldsymbol{a}(v)). \end{aligned} \tag{19.2}$$

Here v is a generic node identifier (the concept of "node" generalizes the concept of discrete time index). The state vector $\boldsymbol{a}(v) \in I\!R^n$ associated with node v is a function of the ordered state vector associated with its children $\boldsymbol{a}(\mathrm{ch}[v]) \in I\!R^{nk}$ and the input vector $\boldsymbol{I}(v)$ that encodes the tuple of attributes attached to node

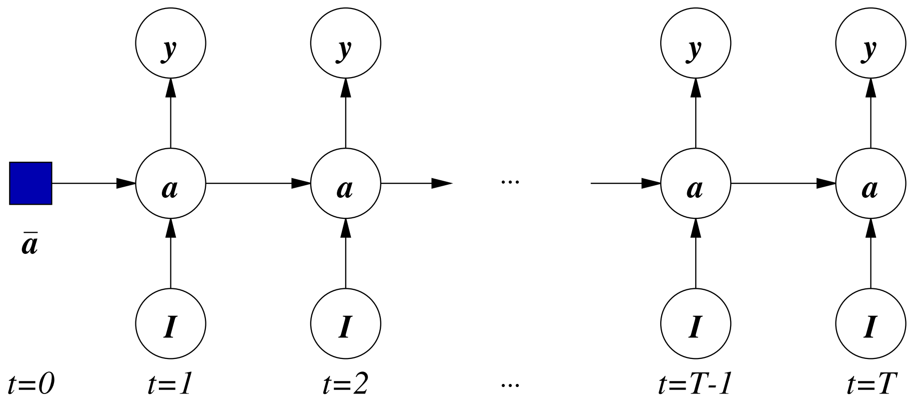

Figure 19.3: Graphical model of the computation carried out by a dynamic neural network on sequences.

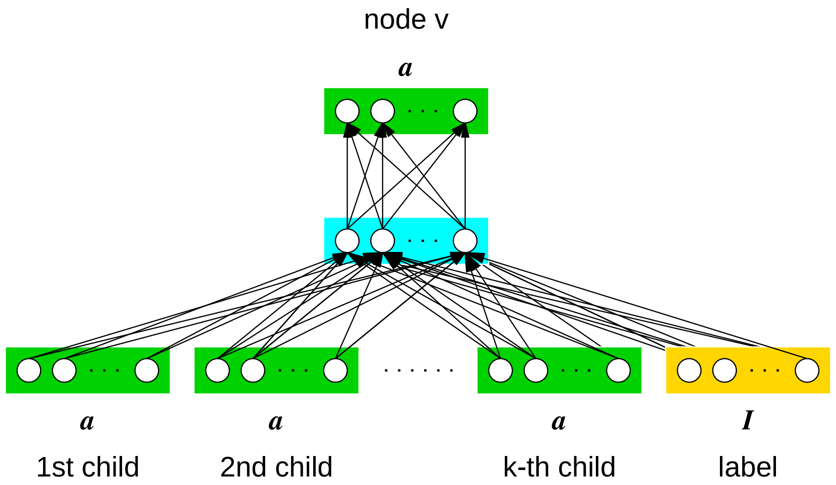

Figure 19.4: A 2-layered MLP implementing the state transition function of a recursive neural network for k-DOAG data structures.

v as a label (see Figure 19.4). The state transition function is assumed to be implemented by a feedforward neural network (for example an MLP, as in Figure 19.4). The base step for recursion in this case corresponds to the absence of children. Whenever a child is missing, the corresponding components of the vector $\boldsymbol{a}(\mathrm{ch}[v])$ are filled from a special frontier state $\overline{\boldsymbol{a}}$ (typically a null vector). The same computational scheme can be applied to k-DPAGs. The only difference is that k-DPAGs may have missing children in any position, whereas missing children in the case of k-DOAGs are necessarily found in the last positions. The second of Equations 19.2 describes the output portion of the model. By means of the function g (also implemented by a feedforward network), the model produces a label $\boldsymbol{y}(v)$ attached to node v in the output graph. In this way, an output graph $\boldsymbol{Y}$, isomorphic to the input graph $\boldsymbol{U}$ is generated. Note that also in this case an explicit dependency on the input can be added and the output can be computed as $\boldsymbol{y}(v) = g(\boldsymbol{a}(v), \boldsymbol{I}(t))$.

19.3.3 Graphical transduction

A deterministic transduction for structured domains is a relation $\mathcal{T} \subset \mathcal{U}^{\#'} \times \mathcal{Y}^{\#}$, where $\mathcal{U}$ and $\mathcal{Y}$ are assigned input and output label spaces, respectively, and $\#$, $\#'$ are assigned skeleton classes. In particular, a deterministic mapping is a function from an input structure space $\mathcal{U}^{\#'}$ to an output structure space $\mathcal{Y}^{\#}$. A probabilistic transduction is described by the joint density $P(\boldsymbol{Y}, \boldsymbol{U})$ of data structures $\boldsymbol{Y}$ and $\boldsymbol{U}$, whose realizations belong to $\mathcal{Y}^{\#}$ and $\mathcal{U}^{\#'}$, respectively. A probabilistic mapping from an input structure space $\mathcal{U}^{\#'}$ to an output structure space $\mathcal{Y}^{\#}$ is described by the conditional density $P(\boldsymbol{Y}|\boldsymbol{U})$.

Learning general transductions of this kind is clearly a formidable task. No connectionist approaches have been proposed so far. The computational model we have introduced only allows implementing special forms of graphical mappings, characterized in the following.

IO-isomorphism

A structured mapping is *IO-isomorphic* if the input and the output data structure have the same skeleton. Using the recursive state-space representation described by Equations 19.2 we can only implement IO-isomorphic mappings. In the special case of sequences, IO-isomorphism means that the input and the output sequences have the same length, i.e. the transduction is *synchronous*. The literature on conectionist models reports only a few attempts to learn asynchronous input-output sequence mappings. Examples of research in this direction are the work by Neco & Forcada [Ñeco and Forcada, 1996] and the work by Bengio & Gingras [Bengio and Frasconi, 1996]. Symbolic approaches have been studied by Oncina *et al.* [Oncina et al., 1993]. At the time of writing, no similar attempts for general data structures have been reported.

Causality

The computational scheme described by Equations 19.2 can be seen as a sequential propagation of information that follows a reversed topological sort of the input graph. Hidden states are in fact updated starting from leaves, toward the supersource. In the case of trees, this form of propagation closely follows the computational scheme employed by frontier-to-root finite automata [Thatcher, 1973]. The main consequence of this message passing scheme is that the output label $\boldsymbol{y}(v)$ cannot depend on input labels other than those found in the subgraph induced by v and its descendants. In the special case of sequences, the same argument translates into the causality property, i.e. the output at time t cannot depend on future inputs.

Stationarity

Both the state transition function and the output function in Equations 19.2 do not depend explicitly on the node v. Temporal dynamics with such a property are called *stationary*. The same terminology is retained for graphical dynamics. The nonstationary extension of the recursive representation of graphical transductions can be defined, in general, as

$$\begin{aligned} \boldsymbol{a}(v) &= f(\boldsymbol{a}(\mathrm{ch}[v]), \boldsymbol{I}(v), v) \\ \boldsymbol{y}(v) &= g(\boldsymbol{a}(v), v). \end{aligned} \tag{19.3}$$

Since f and g are implemented by neural networks (e.g., using MLPs), stationarity can be seen as a particular form of weight sharing. In the case of sequences, this property gives considerable savings in terms of number of free parameters. The same is not necessarily true in the case of other skeleton classes. The structured organization of atoms can be exploited to devise forms of nonstationarity which are not necessarily associated with a dramatic increase in the number of parameters. In the case of trees, for example, weights could be constrained to be the same within the same level, but different weights could be assigned to different levels. In this way, if trees are reasonably balanced, the number of free parameters would only grow as the logarithm of the number of nodes (keeping fixed the size of the networks that implement the transition function).

Supersource transductions

In the case of classification tasks, in which only a categorical variable is needed as output, the output equation becomes more simply

$$\boldsymbol{y} = g(\boldsymbol{a}(s)) \tag{19.4}$$

s being the supersource. In this case, $\boldsymbol{y}$ is a vector of conditional probabilities of categories given the input graph and the network implementing g should have a softmax (normalized exponential) output. Another useful task is regression, in which the input graph is mapped into a continuous (multivariate) variable.

The computational scheme does not change and Equation 19.4 is used for the output. In this case, however, $\boldsymbol{y}$ is interpreted as the expectation of the output variable given the input graph $\boldsymbol{U}$.

19.3.4 On the representational power of recursive models

The representational power of dynamical recurrent networks has been investigated in previous chapters. Let us now briefly summarize and relate the main results that are known in the context of recursive models. As sketched in Section 19.3.2 recursive models for the processing of general data structures can be viewed as the generalization of a large class of dynamical recurrent networks for sequence processing.

As an immediate consequence, recursive models enable universal computation and possess the power of Turing machines and beyond if they are operated according to the setting formulated by Siegelmann and Sontag [Siegelmann and Sontag, 1995, Seigelmann and Sontag, 1994] (the reader is referred to Chapter 9).

The class of finite-state machines that correspond to recursive models with a discrete state representation are known as finite-state *frontier-to-root* (or *bottom-up*) *tree automata* (see Common et al. [Comon et al., 1997], Thatcher [Thatcher, 1973]). Not surprisingly, most concepts, techniques and results developed for the representation of finite-state machines by dynamical recurrent networks (see Chapters 6 and 7) can be applied to represent finite-state frontier-to-root tree automata by recursive network models (see Sperduti [Sperduti, 1997], Gori et al. [Gori et al., 1999] and Küchler [Küchler, 1999a]). These include the "state splitting" technique, the influence of different network architectures (e.g. different number of network layers for the state transition and output mapping, the use of first-order versus second-order activation functions, and threshold transfer functions versus sigmoid transfer functions), and bounds on the "node complexity" of different implementations.

Chapter 8 shows how a certain class of push-down automata can be implemented by dynamical recurrent networks with continuous transfer functions. It is an open question whether and how an appropriate class of *bottom-up tree push-down automata* (see e.g. Coquidé et al. [Coquidé et al., 1994], Common et al. [Comon et al., 1997]) can be represented by recursive network models.

Hammer [Hammer, 1998, Hammer, 1999] investigates the *approximation capability* of recursive network models. It is shown that any measurable function of the type $\mathcal{T} : \mathcal{U}^{\#} \rightarrow I\!R^q$ can by approximated in probability. For a finite input label space $\mathcal{U}$, $\mathcal{O}(\log_2(k))$ MLP layers with $\mathcal{O}(k)$ neurons (first-order activation function and sigmoid transfer function) for each layer suffice for the state transition mapping f. The output function g can be implemented by a two-layer MLP.

19.3.5 Learning Algorithms

Let us now demonstrate how adaptivity and learning is incorporated into models operating on structured domains. The focus is on those models that implement deterministic IO-isomorphic transductions $\mathcal{T} : \mathcal{U}^{\#} \rightarrow \mathcal{Y}^{\#}$ and that are specified by the recursive state representation scheme given with Equation 19.2. The basic idea is to choose parameterized functions f, g for the state transition function and for the output function, respectively, and to estimate the model parameters with regard to the given task and data. We write $\boldsymbol{w}^f$, $\boldsymbol{w}^g$ for the parameters specified for f, g.

Learning as Parameter Optimization

The learning task can often be adequately formulated as the optimization of an *error measure* $E(\cdot)$ which is usually defined on the parameter space $\Omega_f \times \Omega_g$ spread out by the model parameters $\boldsymbol{w}^f, \boldsymbol{w}^g$, and on the outputs computed by the given model (under its current parameter assignment) for data structures $\boldsymbol{U}$ from a given set

$$\mathcal{D} = \{(\boldsymbol{U}, \boldsymbol{Y}), \boldsymbol{U} \in \mathcal{U}^{\#}, \boldsymbol{Y} \in \mathcal{Y}^{\#}\}$$

of input-output examples. This view allows us to attack a number of interesting learning tasks on structured domains, ranging from the supervised classification of graphs, over regression where graphs are mapped to a continuous (multivariate) variable, over the inference of IO-isomorphic transductions, to the optimization of sophisticated criteria defined on top of transductions and on power sets of structures. Examples of relevant tasks and their appropriate formulation by an associated error measure are discussed and put in context of applications in Section 19.4.

Many error measures known for the optimization of static parameterized models appear to be applicable in the context of learning structured information (for a good motivation and a survey of different error functions, we refer to Bishop [Bishop, 1995, Chapter 6]). For the sake of clarity (but without loss of generality) we demonstrate the fundamental mechanisms of learning by means of the popular *sum of squares* (SQE) error measure (Chapter 11).

The error function $E : \Omega_f \times \Omega_g \rightarrow I\!R$, formulated for the inductive learning of IO-isomorphic transductions, accumulates the error contributions of each input-output pair in the data set $\mathcal{D}$. A "naive" translation of the standard SQE error function gives rise to the following definition:

$$E = \sum_{(\boldsymbol{U},\boldsymbol{Y}) \in \mathcal{D}} E((\boldsymbol{U}, \boldsymbol{Y})) = \sum_{(\boldsymbol{U},\boldsymbol{Y}) \in \mathcal{D}} \|\mathcal{T}(\boldsymbol{U}) - \boldsymbol{Y}\|^2. \qquad (19.5)$$

Equation 19.5 has to be refined in order to fit to the structured domain. The "difference in L_2 norm" between an output graph $\mathcal{T}(\boldsymbol{U})$ computed by the model and the target structure $\boldsymbol{Y}$ is rewritten as a standard SQE error formulated on pairs of input and target labels found at corresponding nodes of the given input and target graphs $\boldsymbol{U}, \boldsymbol{Y}$. For the sake of readability, we use the notation for a graph to simultaneously denote the set of its nodes, introduce the function $\boldsymbol{t}(\cdot)$ to access the target label for a given node of the input graph, and drop the output graph parameter $\boldsymbol{Y}$:

$$E((\boldsymbol{U},\boldsymbol{Y})) = E(\boldsymbol{U}) = \sum_{u\in\boldsymbol{U}} E(u) = \sum_{u\in\boldsymbol{U}} \frac{1}{2}\|\boldsymbol{y}(u) - \boldsymbol{t}(u)\|^2. \tag{19.6}$$

Thus, the inductive "learning" of IO-isomorphic transductions on the DOAG domain is translated to

> *find those values for the model parameters $\boldsymbol{w}^f$, $\boldsymbol{w}^g$ that minimize the error function E.*

If the state transition function, f, the output function, g, and the error measure are continuously differentiable functions on the region of interest, then *gradient-based* methods are usually taken into account to solve the optimization problem. Basically, all practical gradient-based optimization methods are built on the local first-order gradient and make use of additional information "between" the first-order and the second-order gradient. We refer to Battiti [Battiti, 1992] for a survey on gradient-based learning methods for feedforward neural networks. The case of dynamical recurrent networks is treated in Chapter 11 (alternatively, see Tsoi [Tsoi, 1998]).

In order to be able to apply gradient-based methods to train models operating on the structured domain (e.g. see Equation 19.2 and Figure 19.4), one has to devise efficient algorithms to compute the first-order gradient information, i.e. $\partial E/\partial \boldsymbol{w}^f$ and $\partial E/\partial \boldsymbol{w}^g$.

ossreference Chapter 13 reviews the two fundamental gradient calculation schemes (and combinations thereof) available for dynamical recurrent networks, namely *backpropagation through time* (BPTT) and *real-time recurrent learning* (RTRL) (alternatively, consult e.g. Williams and Zipser [Williams and Zipser, 1994], Baldi [Baldi, 1995], Pearlmutter [Pearlmutter, 1995]). We argued in Section 19.3.2 that our framework for adaptive processing of structured information can be viewed as a straight-forward generalization of known concepts in the field of dynamical recurrent networks for sequence processing. Not surprisingly, both BPTT and RTRL can be generalized to work on the k-DOAG and k-DPAG domain. Finally, it is worth mentioning that, like any gradient based method, these learning algorithm may suffer problems related to local minima in the error surface. An analysis of this problem can be found in [Frasconi et al., 1997].

Backpropagation Through Structure

Consider the SQE error measure E given by Equation 19.5 and Equation 19.6. The differential of E with respect to a specific model parameter w is written as

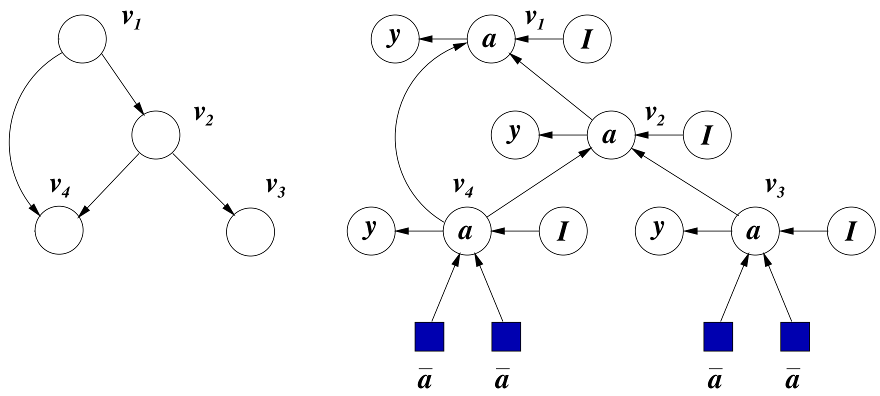

Figure 19.5: Graphical model (right) of the computation carried out by a recursive neural network on an example of 2-DOAG (left).

a sum of differentials for each graph in the data set:

$$\frac{\partial E}{\partial w} = \sum_{(\boldsymbol{U},\boldsymbol{Y})\in\mathcal{D}} \frac{\partial E(\boldsymbol{U})}{\partial w}.$$

The metaphor of *"unfolding in time"*, as used for BPTT, is extended to the *"unfolding through structure"* scheme. Take the functional and recursive state space representation (specified by Equation 19.2) and "unfold" it according to a given input k-DOAG. The result is a graph that is based on the skeleton of the input graph, but the direction of the edges is reversed, each node is augmented by one additional in-going and one out-going edge (to represent the input and the output label, respectively), and missing children are padded with the frontier state $\bar{\mathbf{a}}$ (illustrated by example in Figure 19.5, also cf. Figure 19.3).

Let the parameters occurring in the unfolded model be indexed by the corresponding graph nodes, i.e. we write $\boldsymbol{w}^f(z)$ and $\boldsymbol{w}^g(z)$ to denote the parameter vectors $\boldsymbol{w}^f$ and $\boldsymbol{w}^g$ at node $z \in \boldsymbol{U}$. Thus, the original problem is reduced to the calculation of the partial derivatives of a static multivariate function $E(\boldsymbol{U})$ with respect to the parameters $\boldsymbol{w}^f(z)$ and $\boldsymbol{w}^g(z)$ for $z \in \boldsymbol{U}$. Assuming that the model parameters are *shared* among the nodes, the error gradient of one graph $\boldsymbol{U}$ with respect to the parameter $\boldsymbol{w}^f$ and $\boldsymbol{w}^g$ can be summed up over the individual node contributions.

This rule can be justified by a simple parameter identification applied to the well-known chain rule for partial derivatives. The parameters $\boldsymbol{w}^g(z)$ do only influence the error contribution $E(z)$ of the node z. We get

$$\frac{\partial E(\boldsymbol{U})}{\partial \boldsymbol{w}^g} = \sum_{z\in\boldsymbol{U}} \frac{\partial E(\boldsymbol{U})}{\partial \boldsymbol{w}^g(z)} = \sum_i \frac{\partial E(z)}{\partial \boldsymbol{y}_i(z)} \cdot \frac{\partial \boldsymbol{y}_i(z)}{\partial \boldsymbol{w}^g(z)} = \mathbf{J}^g_w(\boldsymbol{a}(z)) \cdot \boldsymbol{\delta}^g(z), \qquad (19.7)$$

where $\boldsymbol{y}_i(z)$ is the i-th component of the model output computed for node z, and $\mathbf{J}^g_w$ is the *Jacobian* matrix of the output function g with respect to the parameters $\boldsymbol{w}^g$ (which has to be evaluated at location $\boldsymbol{a}(z)$) and $\boldsymbol{\delta}^g$ is given by (see Equation 19.6)

$$\boldsymbol{\delta}^g(z) = \boldsymbol{y}(z) - \boldsymbol{t}(z). \tag{19.8}$$

The error contribution of a node u is influenced by all parameters $\boldsymbol{w}^f(z)$, $z \in \boldsymbol{U}$ where there is a path in the input graph leading from u to nodes z. We have

$$\begin{aligned}\frac{\partial E(\boldsymbol{U})}{\partial \boldsymbol{w}^f} &= \sum_{z\in\boldsymbol{U}} \frac{\partial E(\boldsymbol{U})}{\partial \boldsymbol{w}^f(z)} = \sum_i \frac{\partial E(\boldsymbol{U})}{\partial \boldsymbol{a}_i(z)} \cdot \frac{\partial \boldsymbol{a}_i(z)}{\partial \boldsymbol{w}^f(z)} = \\ &= \mathbf{J}^f_w(\,\boldsymbol{a}(\mathrm{ch}[z]), \boldsymbol{I}(z)\,) \cdot \boldsymbol{\delta}^f(z),\end{aligned} \tag{19.9}$$

where $\boldsymbol{a}_i(z)$ is the i-th component of the state computed for node z, $\mathbf{J}^f_w$ is the Jacobian matrix of the state transition function f with respect to the parameters $\boldsymbol{w}^f$ (which is evaluated at the states $\boldsymbol{a}(\mathrm{ch}[z])$ known for the children of node z and at its input label $\boldsymbol{I}(z)$). The term $\boldsymbol{\delta}^f_i(z)$ is determined as follows:

$$\boldsymbol{\delta}^f_i(z) = \frac{\partial E(\boldsymbol{U})}{\partial \boldsymbol{a}_i(z)} = \sum_j \frac{\partial E(z)}{\partial \boldsymbol{y}_j(z)} \cdot \frac{\partial \boldsymbol{y}_j(z)}{\partial \boldsymbol{a}_i(z)} + \sum_{v\in\mathrm{pa}[z]} \sum_l \frac{\partial E(\boldsymbol{U})}{\partial \boldsymbol{a}_l(v)} \cdot \frac{\partial \boldsymbol{a}_l(v)}{\partial \boldsymbol{a}_i(z)}. \tag{19.10}$$

Equation 19.10 measures the effect of the state $\boldsymbol{a}(z)$ on the error $E(\boldsymbol{U})$. The first term expresses the direct influence on $E(z)$. The second one collects the indirect effects that are propagated through the parents of node z (denoted by $\mathrm{pa}[z]$). To clarify these relationships, consider for example the node v_4 of the input graph depicted in Figure 19.5. The error contribution $E(v_4)$ is directly influenced by the state $\boldsymbol{a}(v_4)$. The error contributions of the nodes v_1 and v_2 are indirectly influenced by $\boldsymbol{a}(v_4)$ via the states $\boldsymbol{a}(v_1)$, $\boldsymbol{a}(v_2)$ and via the state transition function f. Equation 19.10 can be compacted into matrix-vector notation

$$\boldsymbol{\delta}^f(z) = \mathbf{J}^g_x(\boldsymbol{a}(z)) \cdot \boldsymbol{\delta}^g(z) + \sum_{v\in\mathrm{pa}[z]} \mathbf{J}^f_{x(\mathrm{ord}(z,\mathrm{ch[v]}))}(\,\boldsymbol{a}(\mathrm{ch}[v]), \boldsymbol{I}(v)\,) \cdot \boldsymbol{\delta}^f(v), \tag{19.11}$$

where $\mathbf{J}^g_x$ denotes the Jacobian matrix of the output function g with respect to its input arguments and $\mathbf{J}^f_{x(\mathrm{ord}(z,\mathrm{ch}[v]))}$ stands for the transposed Jacobian of the state transition function f with respect to those input arguments that belong to the corresponding child (i.e. $\mathrm{ord}(z,v) = r$ iff the node z is the r-th child of node v).

The gradient calculation scheme given by Equations 19.7–19.11 can directly be translated into an algorithmic specification. Here, we use a pseudo programming language which might typically be found in standard textbooks on algorithmics. Algorithm 1 (Function BPTS) expects a pair $(\boldsymbol{U}, \boldsymbol{Y})$ of input and target graph and returns the first-order gradients with respect to the model parameters $\boldsymbol{w}^g$ and $\boldsymbol{w}^f$. The computation is arranged in two successive *phases*.

During the first phase (Function BPTS-UP), all nodes of the input graph are visited according to a *reverse* topological ordering. To recall, a *topological ordering* for a DOAG is a total ordering of all its nodes such that if the graph contains an edge from node u to node v, then u appears before v in the ordering (for example, v_1, v_2, v_3, v_4 and v_1, v_2, v_4, v_3 are valid topological orderings for the 2-DOAG graph presented in Figure 19.5). This ordering allows us to compute the state for a node $u \in \boldsymbol{U}$ by applying the state transition function f on the input $\boldsymbol{I}(u)$ and on the states $\boldsymbol{a}(\mathrm{ch}[u])$ already computed (compare Equation 19.2 with line 3) and stored for its children (we write $u.\boldsymbol{a}$ to access the state $\boldsymbol{a}(u)$ memorized at the data structure representing the node u).

BPTS($(\boldsymbol{U}, \boldsymbol{Y})$)

1: BPTS-UP($\boldsymbol{U}$)
2: **return** BPTS-DOWN($(\boldsymbol{U}, \boldsymbol{Y})$)

BPTS-UP($\boldsymbol{U}$)

1: Choose a reverse topological ordering $\succ$ on $\boldsymbol{U}$
2: **for** each $u \in \boldsymbol{U}$ obeying the ordering $\succ$ **do**
3:     $u.\boldsymbol{a} \leftarrow f(\mathrm{ch}[u], \boldsymbol{I}(u))$

BPTS-DOWN($(\boldsymbol{U}, \boldsymbol{Y})$)

1: **for** each $u \in \boldsymbol{U}$ **do**
2:     $u.\boldsymbol{\delta}^f \leftarrow \mathbf{0}$
3: $\mathbf{E}^f \leftarrow \mathbf{0}$
4: $\mathbf{E}^g \leftarrow \mathbf{0}$
5: Choose a topological ordering $\prec$ on $\boldsymbol{U}$
6: **for** each $u \in \boldsymbol{U}$ obeying the ordering $\prec$ **do**
7:     $\boldsymbol{\delta}^g \leftarrow \boldsymbol{y}(u.\boldsymbol{a}) - \boldsymbol{t}(u)$
8:     $\mathbf{E}^g \leftarrow \mathbf{E}^g + \mathbf{J}^g_w(u.\boldsymbol{a}) \cdot \boldsymbol{\delta}^g$
9:     $u.\boldsymbol{\delta}^f \leftarrow u.\boldsymbol{\delta}^f + \mathbf{J}^g_x(u.\boldsymbol{a}) \cdot \boldsymbol{\delta}^g$
10:     $\mathbf{E}^f \leftarrow \mathbf{E}^f + \mathbf{J}^f_w(\boldsymbol{a}(\mathrm{ch}[u]), \boldsymbol{I}(u)) \cdot u.\boldsymbol{\delta}^f$
11:     **for** $z \in \mathrm{ch}[u]$ **do**
12:         $r \leftarrow \mathrm{ord}(z, \mathrm{ch}[u])$
13:         $z.\boldsymbol{\delta}^f \leftarrow z.\boldsymbol{\delta}^f + \mathbf{J}^f_{x(r)}(\boldsymbol{a}(\mathrm{ch}[u]), \boldsymbol{I}(u)) \cdot u.\boldsymbol{\delta}^f$
14: **return** $(\mathbf{E}^g, \mathbf{E}^f)$

Algorithm 1: Backpropagation Through Structure (BPTS)

The second phase is realized by Function BPTS-DOWN. The data structure used to store a graph node is augmented by an entry to maintain the $\boldsymbol{\delta}^f$-value. These values and the error gradients are initialized first (lines 1–4). Each node is inspected obeying a chosen topological ordering. This procedure ensures that the $\boldsymbol{\delta}^f$ contributions stemming from the parents have already been summed up when the next node is going to be visited. Thus, Equation 19.11 is implemented by line 9. The error gradient contributions of the node under consideration are accumulated to the overall error gradients (lines 8,10). Finally, the $\boldsymbol{\delta}^f$-value is ***propagated back*** ("the name of the game") onto its children and is added to those values already found there (lines 11-13).

Let N_E (and $N_V = |\boldsymbol{U}|$) be the number of edges (and nodes) contained in a given input k-DOAG $\boldsymbol{U}$. Assume that the underlying recursive model is specified by a state transition function f and by an output function g of the type

$$f : \mathbb{R}^{l+kn} \to \mathbb{R}^n, \qquad g : \mathbb{R}^n \to \mathbb{R}^q,$$

i.e. the input (output) labels are of dimensionality l (q) and the state space is contained in $\mathbb{R}^n$. Furthermore, let f and g be configured by $N_f = \dim(\Omega_f)$ and $N_g = \dim(\Omega_g)$ parameters.

Neglecting the space to maintain the input graph, the space requirements of the BPTS algorithm (Algorithm 1) are characterized by

$$\Theta(N_V n + N_f + N_g).$$

The computational effort in time is determined by N_V evaluations of the state transition function f (for the BPTS-UP phase) and the effort induced by the BPTS-DOWN phase. The later is given by N_V matrix-vector multiplications involving the Jacobians $\mathbf{J}_w^g \in \mathbb{R}^{N_g \times q}$ (line 8), $\mathbf{J}_x^g \in \mathbb{R}^{n \times q}$ (line 9), $\mathbf{J}_w^f \in \mathbb{R}^{N_f \times n}$ (line 10) and N_E matrix-vector multiplications concerning the Jacobian $\mathbf{J}_{x(\cdot)}^f \in \mathbb{R}^{n \times n}$ (line 13), yielding

$$\Theta\big(N_V(N_g q + nq + N_f n) + N_E n^2\big)$$

multiplications. Additionally, the effort for the evaluation of the corresponding Jacobians must be taken into consideration. The asymptotic time behavior of BPTS is not affected by a topological sort of the DOAG nodes, which can be accomplished in $\Theta(N_V + N_E)$ time.

Note that a simple trick can be used to avoid the explicit evaluation of the Jacobians. View both the state transition function, f, and the output function, g, as static feedforward networks that are composed of elementary functions. Doing so, the $\boldsymbol{\delta}^g$ and $\boldsymbol{\delta}^f$ values can be propagated back through the graph and for each node back through the functional networks g and f. For a formal proof and a detailed discussion of this topic we refer to Wan and Beaufays [Wan and Beaufays, 1998].

For example, suppose that each of the transition function f and the output function g were implemented by an one-layer MLP (with full connectivity to the inputs). Thus, the space requirements are given by

$$\Theta\big(N_V n + n(l + kn) + nq\big).$$

The BPTS-UP phase corresponds to one forward pass and the BPTS-DOWN phase to one backward pass of the standard backpropagation algorithm through the network unfolded according the given input graph. The computational effort in time is given by

$$\Theta\big(N_V(nq + n(l + kn)) + N_E n^2\big).$$

Note that in this example BPTS is reduced to BPTT for the *simple recurrent network* (Chapters 11) if the structured domain is constrained to sequences (k-DOAGs with $k = 1$).

Some Comments on the Historical Roots of BPTS

As with BPTT (Chapter 11), BPTS has probably been re-discovered several times in the past. The form and usage presented here is due to Goller and Küchler [Goller and Küchler, 1996, Küchler and Goller, 1996], who developed the backpropagation through structure gradient calculation scheme for recursive network models (the state transition function f and the output function g are implemented by MLP's) to infer and to represent supersource transductions (compare Equations 19.4) on the DOAG domain.

However, the roots of BPTS can presumably be traced back to the development of connectionist models for the representation and processing of tree structures during the early nineties. These models consist of coupled sub-modules, e.g. RAAM's (see Section 19.2) and feedforward architectures, and were trained by gradient-based optimization methods (see Chrisman [Chrisman, 1991]). Although not explicitly worked out in the literature, the backpropagation scheme has been utilized in a recursive way to compute first-order gradient contributions for tree structures (or parts thereof) with respect to the encoder and decoder of the RAAM modules. Stolcke and Wu [Stolcke and Wu, 1992] proposed a "parallel training" mode where *"... the error between the RAAM-encoded result and the output of the unification network is backpropagated both through the unification network and also recursively through the RAAM network ... "*. Berg [Berg, 1992] developed a connectionist parser to map sequences of words (sentences) to tree structures (the corresponding syntactical parse tree). The architecture is built of two components: a simple recurrent network and the decoder part of the RAAM model. The training procedure is described in a verbal form *"... the network used in training is actually a number of virtual copies of the decoder portion of the parsing network ... This technique is similar in form, if not in purpose, to back-propagation through time ... "*.

RTRL on structured domains

The RTRL gradient computation scheme for dynamical recurrent networks operating on temporal information is characterized by *locality in time*, but it lacks locality in space (see Chapter 11). Due to its locality in time, RTRL allows to devise *online* update rules for adapting dynamical recurrent networks to an *infinite* stream of in-going data. BPTT is global in time and therefore not applicable to infinite sequences. Let us briefly investigate the situation when RTRL is applied on the structured domain (introduced by Sperduti and Starita [Sperduti and Starita, 1997]).

Again, the overall error gradient for a given input graph $\boldsymbol{U}$ is given by the sum of the error gradient contributions of its nodes:

$$\frac{\partial E(\boldsymbol{U})}{\partial w} = \sum_{u \in \boldsymbol{U}} \frac{\partial E(u)}{\partial w} = \sum_{u \in \boldsymbol{U}} \sum_i (\boldsymbol{y}_i(u) - t_i(u)) \cdot \frac{\partial \boldsymbol{y}_i(u)}{\partial w}. \tag{19.12}$$

The error gradient of one node with respect to the model parameters found in the output function g is immediately given by

$$\frac{\partial E(u)}{\partial \boldsymbol{w}^g} = \mathbf{J}_w^g(\boldsymbol{a}(u)) \cdot \boldsymbol{\delta}^g(u). \tag{19.13}$$

By applying the well-known generalized chain rule for differentials on the error contribution of one node with respect to the parameters $\boldsymbol{w}^f$, we get

$$\frac{\partial E(u)}{\partial \boldsymbol{w}^f} = \mathbf{P}(u) \cdot \big(\mathbf{J}_x^g(\boldsymbol{a}(u)) \cdot \boldsymbol{\delta}^g(u)\big), \quad \text{where} \quad P_{ij}(u) := \frac{\partial \boldsymbol{a}_j(u)}{\partial \boldsymbol{w}^f}. \tag{19.14}$$

The recursive state dynamics (specified by Equation 19.2) can be utilized to directly evolve the partial derivative $\mathbf{P}(u) \in I\!R^{N_f \times n}$ to

$$\mathbf{P}(u) = \mathbf{J}_w^f(\boldsymbol{a}(\mathrm{ch}[u])) + \sum_{z \in \mathrm{ch}[u]} \mathbf{P}(z) \cdot \mathbf{J}^f_{x(\mathrm{ord}(z,\mathrm{ch}[u]))}(\,\boldsymbol{a}(\mathrm{ch}[u]), \boldsymbol{I}(u)\,). \tag{19.15}$$

Equation 19.15 measures the total impact of the parameters $\boldsymbol{w}^f$ on the target $\boldsymbol{a}(\cdot)$, i.e. the direct effects ($\mathbf{J}_w^f$, via f) and the indirect effects (via the recursive composition of f on the children of a node). One can easily verify that the recursion is well-founded.

The computation of the state information $\boldsymbol{a}(u)$ can be interwoven with the calculation of the matrix $\mathbf{P}(u)$. Only one pass through the nodes u of a given input graph $\boldsymbol{U}$ in a reverse topological ordering is required. The computational costs in time are determined by N_V matrix-vector products involving the Jacobians $\mathbf{J}_w^g$ (Equation 19.13), $\mathbf{J}_x^g$ (Equation 19.14), and by N_E matrix-matrix products (Equation 19.15), yielding

$$\Theta\big(N_V(N_g q + nq) + N_E N_f n^2\big)$$

multiplications. If the state transition function f and the output function g are specified themselves as complex functions, then (in contrast to BPTS) there is no coherent way to avoid the evaluation of the Jacobians. However, the i-th row of a Jacobian can alternatively be calculated by using the backpropagation scheme to propagate the i-th unit vector back into the function network.

Thus, concerning time, BPTS is clearly *superior* to RTRL. Consider the neural implementation (one-layer MLP for each of f and g) introduced as example at the end of Section 19.3.5. The time complexity of RTRL is given by

$$\Theta\big(N_V n q^2 + N_E n^3(l + kn)\big).$$

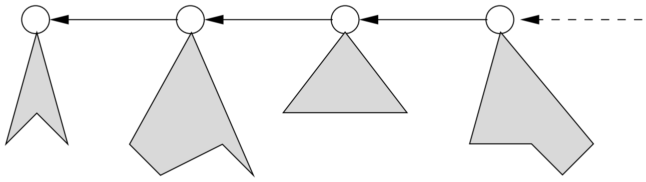

Figure 19.6: Illustration of the "never-ending movie" input skeleton: DOAGs of an arbitrary, but finite skeleton (depicted by grey-shaded areas) are glued to the nodes of an infinite sequence.

Stated in loose terms, the computational effort of BTPS in time scales *quadratically* to the number of neurons used to represent the state space and *linearly* in the number of edges ($\mathcal{O}(N_E k n^2)$), while that of RTRL is growing with $\mathcal{O}(N_E k n^4)$. The space requirements of RTRL for structures ($k > 1$) are rather demanding. One can easily devise an input DOAG where, due to the dependencies between nodes and the recursive state dynamics, $\Theta(N_V)$ times memory blocks, equivalent to store the matrix $\mathbf{P}$, have to be allocated (see Küchler [Küchler, 1999b]) . Thus, we get a worst case memory consumption of

$$\mathcal{O}(N_V N_f n + N_g).$$

If one is considering only tree domains (i.e. DOAGs, where each node has at most one in-going edge), the space consumption is given by $\mathcal{O}(kdN_f n + N_g)$ where d is denoting the depth of the input tree. This means that RTRL is *loosing* its locality in time (corresponding to a node-locality in our framework) when applied on k-DOAG domains with $k > 1$, i.e. the space consumption *cannot* be kept independent of the size (number of nodes) of the input structure. For RTRL operating on sequences ($k = 1$), it suffices to maintain two matrices of the type $\mathbf{P}$ and to use them in an alternating way to compute $\mathbf{P}(t+1)$ from the information found in $\mathbf{P}(t)$ at any point t in time.

Even if the memory consumption is a matter of concern, there are only very few scenarios where RTRL for structures ($k > 1$) is to be favored to BPTS. One example is constituted if the input is given by *very large* trees ($N_V \gg N_f kd$). Note that there is no reasonable definition for recursive models to process infinite input structures of a general DOAG skeleton. Moreover, the reader is reminded that RTRL would require infinite memory in the worst case.

Figure 19.6 illustrates a very exotic example of an infinite input skeleton ($k > 1$) that could be processed with recursive models, using RTRL in *node-online* parameter update mode, if some modifications were made to the core algorithms. The skeleton of this structure might be best explained by a "never-ending movie" where each image is represented by an arbitrary DOAG structure of finite size. Suppose that a recursive model is trained to recognize properties of the movie, e.g. whether it is interrupted by commercials during the broadcasting, at any point of time. For certain periods of time, explicit target values are provided and the system is trained in a supervised manner. Thus, the target skeleton is constituted by infinite sequence.

In order to be applicable to this movie scenario, the RTRL algorithm has to undergo a major modification. Firstly, the image graphs have to be processed in a left-to-right order. Furthermore, an explicit memory management has to be employed to limit the space consumption to a finite amount. The parameter update is performed every time t the gradient information $\mathbf{P}(t)$ has been computed for a supersource node of an image representation. Afterwards, the memory allocated for the tree at time $t-1$ can be freed again.

19.4 Applications

Adaptive structure processing allows one to open new interesting application domains. Specifically, it can be employed in all application domains where a *quantitative* evaluation of structured information is required, but no procedural evaluation can be precisely defined. The quantitative evaluation of structures is often introduced *to master the complexity of the problem*, i.e., to reduce the complexity of the search space to a manageable size. This implicitly means that any useful realization of the evaluation function must be *robust* to "noise"in the structures and to mislabeling of training examples, i.e., a form of *continuity* of the evaluation function on the structured domain is assumed. In some applications, the input structures may present *both categorical and numerical data*, e.g., in chemistry, and both types of information are believed to be necessary for the determination of the desired output. Finally, a *parsimonious realization* is required, otherwise the requirement for conciseness, and thus for efficiency, would not be satisfied. Note that, in the context of inductive learning, the requirement for a parsimonious realization is also believed to have beneficial effects on generalization.

All the above considerations fit well the computational capabilities of adaptive structure processing. In this section, we present some fascinating directions that have been explored (and are under investigation) by several research groups.

19.4.1 Pattern Recognition

During the 70's and 80's the field of pattern recognition has been significantly influenced by the school of "Syntactical and Structural Pattern Recognition" [Pavlidis, 1977, Gonzalez and Thomason, 1978, Schalhoff, 1992] which is based on the premise that the *structure* of an entity is very important, both for classification and description. Despite the common point of view, syntactic and structural pattern recognition have a precise characterization: syntactic pattern recognition uses tools from the theory of formal languages, while structural pattern recognition is mainly based on graph matching.

The necessity of such a structural point of view became clear when researchers in traditional pattern recognition realized that feature derivation is a very complex process that can hardly be automated and that features can be both numerical and symbolic. Moreover, it was observed that in several application domains low-level features can naturally be organized into higher-level structures. These structures are invariant with respect to some transformations, such as translation, scale, rotation, relative displacement, and so on, and can be recognized by a matching with reference models.

Syntactic and structural techniques have successfully been applied in a variety of problems, mainly originated from *document analysis*. Examples of applications are: processing of two-dimensional mathematical notations [Belaïd and Haton, 1984], analysis and recognition of printed documents [Nagy et al., 1992, Green and Krishnamoorthy, 1996], representation of printed music [Baumann, 1995], analysis of diagrams [Groen et al., 1985, Lee, 1992], and automatic extraction of information from engineering drawings [Collin and Colnet, 1994, Dori, 1989].

A wider application of these approaches, however, is limited by some still unsolved problems like sensitivity to noise and errors, model inference, descriptive power [Tombre, 1996]. Noise is usually introduced into data by scanning, discretization, and image processing, as well as by the overlapping of several information layers, e.g., in engineering drawings text strings can touch lines. The sensitivity of the structural approach to noise is due to the application of a parser for the extraction of structures from the processed data: if the intersection of two lines in a drawing is lost, then also structural information is lost and the output of the parser may be very different. Also the automatic selection of relevant portions of the data is a major problem, since a brute force search for all the possible structural aspects present in the data is infeasible. This last aspect is related to the lack of tools for efficient inference of structural information which may include both categorical and numerical variables. As a major drawback, the lack of these tools renders it very difficult to apply the approach to complex problems which may involve hundreds of thousands of items. Finally, grammars can represent only a limited type of knowledge, especially when the interaction between categorical and numerical entities must be considered.

Adaptive structure processing allows us to attack these limitations and suggests a new way of pattern recognition in graphical domains. First of all, the representation of structural information can be integrated very naturally with numeric information. Then, the models proposed in this chapter integrate the parsing process with the inference process. In fact, the encoding scheme discussed in Section 19.3 is devised on the basis of a training set and its functional scheme is adaptive, depending on the specific classification or approximation task at hand. So, even if noise is present in training data, the inference process is able to balance errors of the parsing process. Finally, the result on the universal capability of this class of models guarantees that, given a desired function, there exists a network able to realize it.

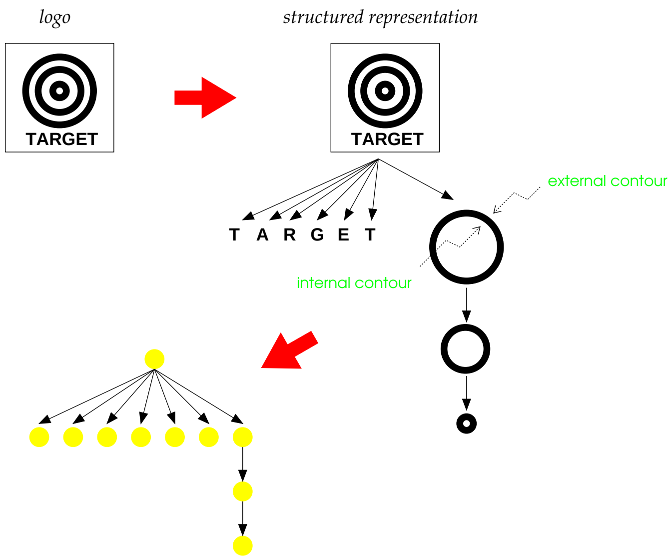

Figure 19.7: Example of how a logo can be represented as a structure. Specifically, the logo on the left side of the picture can be represented by a tree where the different parts of the logo are disposed according to concept "included in". This tree representation is then completed by attaching to each node of the tree a label reporting local features of the parts, like, for example, the area or the perimetral of the internal and/or external contour of the part. Other more sophisticated features can be included.

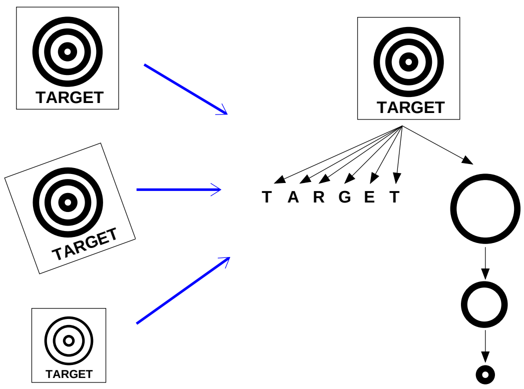

Figure 19.8: The tree representation is particularly convenient when considering global transformations of the logo, like scaling and rotations. In fact, the structured representation is invariant with respect to these transformations.

19.4.2 Computational Chemistry

Computational Chemistry/Biology is another very interesting application field for adaptive structure processing. For instance, in drug design, Quantitative Structure Activity Relationships (QSARs) are of specific interest. The problem is to relate (or map) the structure of a chemical compound to its associated chemical activities (usually represented summarily by a scalar continuous value). Typically, chemical compounds are given by two-dimensional (or three-dimensional) graph representations; vertices correspond to atoms and edges to bonds between atoms. Since there exists no "closed universal theory" on drug synthesis, in order to reduce the costs and speed the drug design processes, one would like to have a method to guide the prediction of the activity of new synthesized compounds utilizing knowledge on compounds that already have been analyzed in laboratories and that is archived in databases. The aim is to discover which substructure or which set of substructures characterize the biomechanism of activity. When considering different methods, the most relevant aspect, which characterizes one method with respect to another one, is the way the molecules are represented. Different techniques to represent the structure of a drug have been developed, such as *molecular graphs*, *structural elements*, and *topological indices*. Topological indices, in particular, represent features of the molecule structure in numerical form. They are suitable for use in structure-activity relationships that may be developed by standard regression methods, or as basis for pattern recognition and discriminant analysis. Feed-forward neural networks have been applied with different modalities to QSPR and QSAR [Zupan and Gasteiger, 1993]. However, just as traditional QSPR/QSAR approaches, the standard approach with feedforward neural networks consists of encoding each graph as a fixed-size vector of features, which is then used for feeding the neural network. The features used may involve physical-chemical parameters [Ajay, 1993], or topological indexes, or vectorial graph codes [Cherqaoui and Villemin, 1994]. Unfortunately, since the structural features which are relevant for the task at hand are unknown, all these representations are defined through a very expensive trial and error approach. On the contrary, the adaptive structure processing framework constitutes a computational environment where the relevant representations for the molecular structure is devised by the encoding phase on the basis of a set of examples, i.e., a set of molecules for which the desired activity is known. Moreover, in addition to the information on the molecular structure, it can process uniformly both categorical data, i.e., the type of atoms, and numerical data, i.e., physical properties of the bonds.

The main requirement for the use of the adaptive structure processing framework consists in finding a representation of molecular structures in terms of DOAGs. The candidate representation should retain the detailed information about the structure of the compound, atom types, bond multiplicity, chemical functionalities, and finally a good similarity with the representations usually adopted in chemistry. As previously discussed, the main representational problems are how to represent cycles, how to give a direction to edges, and how to define a total order over edges. According to the family of chemical compounds and computational tasks at hand, different solutions to these problems can be devised. Some common guidelines, however, can be individuated. For example, let us consider the problem of how to represent cycles in chemical compounds. In many QSAR problems, family of compounds sharing a common template are studied. Very often cycles are just present in the common template and so there is no need to explicitly represent cycles, i.e., a single node representing the common template can be used. If this is not the case, cycles can be treated as entities at a higher level of abstraction and special labels, characterizing the chemical nature of these molecular sub-groups, can be defined.

The problem of giving a direction to edges can, in general, be solved by resorting to the definition of a set of rules based on the I.U.P.A.C. nomenclature system, which establishes how a chemical formula should be written in linear form. This solution is straightforward when a supersource is already defined, however, it can itself be used in order to select a supersource. Finally, the total order over edges is usually defined by a set of rules mainly based on the size of the sub-compounds.

Specific instances of these general rules have been applied in [Bianucci et al., 1999b] to represent Benzodiazepines and Alkanes as labeled trees to perform a QSAR and a QSPR study, respectively, by Cascade-Correlation networks for structures. The QSAR task was the prediction of the non-specific activity (affinity) towards the Benzodiazepine/$GABA_A$ receptor by a group of Benzodiazepines (Bz) [Hadjipavlou-Litina and Hansch, 1994], while the QSPR task consisted in the prediction of the boiling point for a group of acyclic hydrocarbons (Alkanes)[Cherqaoui and Villemin, 1994]. The above work demonstrated that the application of neural networks for structures to QSAR/QSPR tasks allows the treatment of different computational tasks by using the same basic representations for chemical compounds, obtaining improved prediction results with respect to traditional equational approaches for QSAR and competitive results with respect to 'ad hoc' designed representations and MLP networks in QSPR. It must be stressed that the possibility of representing a chemical compound by a structured representation is very important since it is possible to store into the label attached to each node information at different levels of abstraction, such as the atom types or functional groups, allowing a flexible treatment of different aspects of the chemical functionality. Concerning a comparison with respect to approaches based on standard feedforward networks, the main advantage resides in the fact that the encoding of chemical structures does not depend on a fixed vector- or template-based representation. In fact, due to the dynamical nature of the computational model, neural networks for structures are able to adapt the encoding process to the specific morphology of each single compound, as demonstrated by the results obtained by the study of the internal representations developed by the Cascade-Correlation networks used for the QSAR/QSPR tasks [Bianucci et al., 1999a]. As a final comment, it must be observed that the application of neural networks for structures to chemistry seems to be very promising since it is able to address computational problems which still are not reachable from traditional symbol-based systems, such as Inductive Logic Programming (ILP)[Muggleton and Raedt, 1994]. In fact, even if ILP has been successfully applied to some problems in chemistry, the prediction of a numerical value from the structure of the compound does not seem to be possible within the ILP framework.

19.4.3 Adaptive Search Control for Symbolic Deduction Systems

The following scenario describes the challenging task of attempting to improve conventional symbolic problem solvers, e.g. *automated deduction systems*. Automated deduction systems are general-purpose tools for automatically deriving formal proofs for conjectures based on a set of axioms (see Wos et al. [Wos et al., 1992, Wos, 1990]). Both conjectures and axioms are expressed in a formal language, for example by first order predicate logic, exploiting *declarative knowledge* on the application domain.

Applications can be found in such fields as logic programming, computer algebra, expert or knowledge-based systems, planning, deductive databases, formal software and hardware verification, etc.

The central problem in automated deduction is the explosive growth of *search spaces* when the proof length increases. Clearly, this is a price that must be paid for the ability to process declarative and expressive formalisms. Many interesting proof tasks induce *infinite* search spaces. Therefore, most systems make use of *"heuristics"* (i.e. clever navigation strategies) to control the proof search, whenever this is possible. However, there appears to be a certain trade-off between the generality of heuristics and their effectivity. Heuristics which are shown to be very effective for one application domain, sometimes turn out to be counter-productive for other domains and proof tasks. Moreover, in many application domains there are no such heuristics known in advance, and it is very difficult and time-consuming to derive them by hand.

One recognized way out of this dilemma is to exploit past proof experience. The idea is to use feasible proof tasks from one given domain, which can either be solved by brute force search or by the application of standard heuristics, and to automatically *learn heuristics from data* that is extracted from the search spaces explored so far. The hope is that the acquired heuristics generalize to unseen but related proof tasks, enabling an acceleration of the search process and the solution of proof tasks that were not reachable before. Search-control heuristics could be represented as functions that compute *numeric ratings* for alternative states or inference steps. However, all relevant state and context information is "hidden" in logical formulae and functional terms, i.e. in rooted ordered trees $\mathcal{U}^{\#}$ over an alphabet $\mathcal{U}$. Furthermore, it does not suffice to consider isolated states during the learning process. Rather, information has to be integrated over several proof (and non-proof) paths and also collected among several proof tasks.

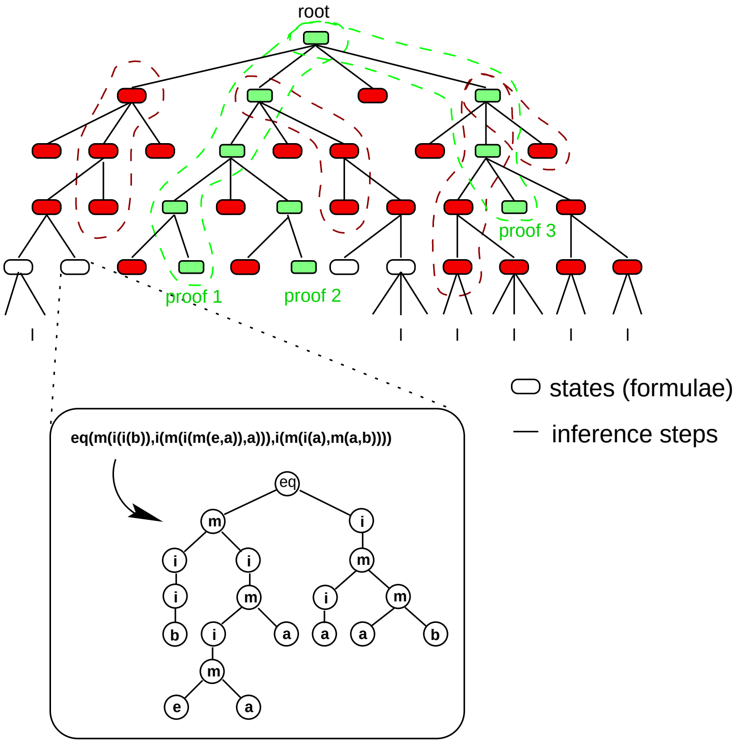

Figure 19.9: A finite initial segment of a search-tree generated by an automated deduction system for one specific proof problem. Light grey boxes denote states lying on solution paths; explored states that did not lead to proofs are marked by boxes in dark grey. In order to simplify this drawing, only some of the occurring solution-paths and some of the failure-paths have been illustrated by surrounding dashed lines drawn in light grey and dashed lines in dark grey, respectively. States are characterized by logical formulae, which in turn can be viewed as tree structures labeled by symbols. The semantics of the symbols are specified by the underlying logic and by the logical axiomization of the problem to be solved.

This situation is sketched by Figure 19.9. The search space has the shape of a tree and is explored, starting from the root, until the first proof is found. If more than one inference rule is applicable, an effective heuristic has to choose, from among the several alternatives, the one leading to a short solution path (states marked in light grey). Grey boxes denote states already visited while the white ones stand for unexplored states, both possibly rooted at infinite parts of the search space.

Goller [Goller, 1997, Goller, 1999] develops a novel framework for the inference of search-control heuristics for automatic deduction systems. Roughly stated, the idea is to infer, exploiting past proof experiences, an evaluation function

$$eval : \mathcal{U}^{\#} \to (-1, +1)$$

that enables the selection of the most promising states (or inference steps) among several alternatives during the proof search for unseen but related proof problems. Without loss of generality, it is assumed that high ratings are characterizing good choices. The search strategy globally compares ratings. A state s_i is explored before state s_j if

$$\exists\, s'_j \in pred(s_j)\ \forall s'_i \in pred(s_i) : eval(s'_i) > eval(s'_j),$$

where *pred* is the reflexive and transitive closure of the immediate predecessor relation in a search-tree.

A recursive neural network model is used to represent the evaluation function *eval*. There are no explicit target ratings for the states in the training data. The error criterion combines state evaluations from all *solution-paths* (L^+) and *failure-paths* (L^-) of a search-tree (compare Figure 19.9). A solution-path is the set of all states lying on a path from the root of the selected initial segment of the search-tree to a proof. *Negative examples* are such states that are explored in the training sample, but are not on a solution-path. Each negative example which does not have a successor state within the sample defines a failure-path. This failure-path contains the negative example itself, and all its predecessor states that are not members of all solution-paths. Goller [Goller, 1997] proposes an error criterion that aims at satisfying the following inequality:

$$\max\{\min\{eval(s)\}_{s \in l}\}_{l \in L^+} > \max\{\min\{eval(s)\}_{s \in l}\}_{l \in L^-}$$

This means that all states on a solution-path should get a high rating. However, it suffices if there exists *one* solution path where each state is ranked higher than all other states on failure-paths. This error criterion is easily extended to combine samples from several proof problems. The min and max operators are approximated by continuously differentiable functions. Thus, the overall error criterion can be optimized (in the parameter space of the embedded recursive model) by gradient-based optimization methods built on top of the BPTS algorithm (see Section 19.3.5 and Algorithm 1).

The feasibility of this approach is demonstrated (see Goller [Goller, 1999]) by means of the general-purpose automated deduction system SETHEO (see Letz et al. [Letz et al., 1992], and Moser et al. [Moser et al., 1997]) and with the help of experiments in the domain of *word problems in group theory* (WPGT). A WPGT consists in answering the question whether two ground terms are equal with regards to group theory. 317 different WPGT were randomly generated. 298 of these proof problems were solvable by SETHEO using the built-in iterative deepening search strategy with inference bound and a time limit of $80s$. The length of the proofs varied between one and eleven inference steps. The recursive neural network models were trained on the data extracted from the search-trees that had been induced by the 298 solvable proof problems.

The performance of the SETHEO system in combination with the trained recursive network as search control module has been tested on the remaining 19 proof problems, which were intractable for the original system. Again a time limit of $80s$ (inclusively the time for the search control) was used. 16 of these test problems could be solved with an average time of $2.9s$. E-SETHEO (Moser et al. [Moser et al., 1997]), a version of SETHEO with special equality handling mechanisms, was able to solve 17 test problems within the time limit. However, the average search time was $5.5s$. These results give some evidence that the proposed approach is capable to inductively infer reasonable search control heuristics. The acquired heuristics appear to generalize (to a certain extent) to novel proof problems from the same problem domain.

Note that the proposed approach is rather general and not restricted to automated deduction systems in the classical sense. There is no reason why other symbolic reasoning systems, for example planning- and computer algebra systems, deductive databases, expert systems, etc., should not profit from the proposed idea to adapt the search control to the problem domain under consideration.

Historical Remarks

A first attempt to learn an evaluation function in SETHEO was performed by training a back-propagation network on feature vectors representing predefined features for the structures encountered during the search, like the proof-context and the applicable inference steps [Suttner and Ertel, 1991]. Examples of *static* features, i.e., features which can be computed at compilation time, are the number of literals in a clause, or the number of distinct variables in the head of a clause; examples of *uniform dynamical* features, i.e., features computed at run time but uniform for all branches of an **or** branch-point, are the current depth in the search tree or the number of instantiated variables in the calling subgoal; finally, there are *distinct dynamical* features, i.e., which depend on the actual clause occurrence at run time, like the current total number of uses of a given clause in the current search or the number of variables of a calling clause becoming instantiated[‡]. The results obtained with this network and other standard neural networks models [Goller and Küchler, 1996] improved the search time of one order of magnitude, however, as is clear from the examples of features given above, the encoding of structural information in a fixed-size vector is not able to capture all the relevant information gathered by symbolic structures of arbitrary size. In fact, it is important to include the selection of the most relevant features as part of the learning task. A preliminary move towards this direction is reported in [Sperduti, 1995], where LRAAM-based networks were successfully used to perform classification of symbolic recursive structures encoding logical terms. A refinement of this work led to the definition of the backpropagation through structure algorithm [Goller and Küchler, 1996]. The

[‡]Note that this feature is different from the number of instantiated variables in the calling subgoal, since the subgoal is the same for each clause for which unification is tried.

experimental comparison between the LRAAM-based approach and the backpropagation through structure algorithm reported in [Goller and Küchler, 1996] shows that the latter algorithm obtains slightly better results for all examples solvable by the LRAAM-based approach, but with smaller networks and training times. Moreover, the backpropagation through structure algorithm was able to successfully learn classification tasks that could not be solved in a reasonable amount of time by LRAAM-based networks. Another improvement in this direction has been the development of the Cascade-Correlation Network for structure [Sperduti et al., 1996, Sperduti and Starita, 1997] (a generalization of recurrent Cascade-Correlation for sequences [Fahlman, 1991]) which obtained better results with respect to the LRAAM-based networks on a subset of the classification problems reported in [Sperduti et al., 1995]. One advantage of the Cascade-Correlation Network for structure over backpropagation through structure networks is that the necessary number of hidden units is automatically determined by the learning algorithm.

19.5 Conclusions

The framework discussed in this chapter opens the doors to the adaptive processing of information, where different pieces are related by means of directed acyclic graphs. Our formalism can be intuitively explained by extending models for serially ordered sequences to models capable of dealing with partial orders among the variable described by DOAGs. Therefore, it seems quite natural to expect that many theoretical analyses for recurrent networks and HMMs can be generalized to the recursive models presented in this chapter. In particular, one interesting issue is the introduction of symbolic prior knowledge into adaptive structural processors. Several prior knowledge injection algorithms have been introduced for recurrent networks, assuming that the available prior knowledge can be expressed in terms of state transition rules for a finite automaton (see Chapter 10). Because of the strict relations between recurrent and recursive networks pointed out in this chapter, most of the methods adapted to incorporate and extract symbolic knowledge for the case of linear lists are likely to be extended to the case of graphs. For instance, the natural relationship between automata and second-order recurrent neural networks can be extended to a relationship between m-ary tree automata and $(m+1)$-th order recursive networks. Unlike recurrent networks used to process sequences where the partial specification of the transduction can only involve the information attached to the nodes of the list, in the case of recursive networks, the graph topology can be an additional rich source of prior knowledge. Because of the hybrid nature of the data, there are cases in which the graph topology can itself be used to create a partition of the training set. The prior knowledge on the graph topology can be exploited for designing a *modular recursive network*, in which the single modules are learned separately on the basis of the partition of the training set operated by the prior topology information.

The applications shown in this chapter are very promising and indicate clearly the potential features of the proposed approach. Interestingly enough, other classical disciplines like syntactic and structural pattern recognition rely on very related methodologies. The strength of the proposed approach seems to be mainly related to the natural way of representing objects with either numerical or symbolic information. Many applications faced with traditional connectionist models are likely to benefit from the proposed approach. However, it seems very reasonable to expect a significant benefit from those cases in which the examples exhibit a significant structure.