Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 344 (2005) 298–334
Theoretical Computer Science
www.elsevier.com/locate/tcs

From Hopfield nets to recursive networks to graph machines: Numerical machine learning for structured data

Aurélie Goulon-Sigwalt-Abram[a], Arthur Duprat[b], Gérard Dreyfus[a,*]

[a] *Laboratoire d'Électronique, ESPCI-Paristech, (CNRS UMR 7084), 10, rue Vauquelin, 75005 Paris, France*
[b] *Laboratoire de Chimie Organique, ESPCI-Paristech, (CNRS UMR 7084), 10, rue Vauquelin, 75005 Paris, France*

Received 26 July 2005; accepted 8 August 2005

Communicated by C. Torras

Abstract

The present paper is a short survey of the development of numerical learning from structured data, an old problem that was first addressed by the end of the years 1980, and has recently undergone exciting developments, both from a theoretical point of view and for applications. Traditionally, numerical machine learning deals with unstructured data, in the form of vectors: neural networks, graphical models, support vector machines, handle vectors of features that are assumed to be relevant for solving the problem at hand (classification or regression). It is often the case, however, that data is structured, i.e. is in the form of graphs; three examples will be described here: prediction of the properties of molecules, image analysis, and natural language processing. The traditional approach consists in handcrafting a vector representation of the structured data (features describing the molecules, "bag of words" for language processing), and subsequently training a machine to perform the task from that representation. By contrast, we describe here a family of approaches (RAAMs, LRAAMs, recursive or folding networks, graph machines) that are specifically designed to learn from structured data. We show that, despite the apparent diversity, two basic principles underlie the recent approaches: first, use structured machines to learn structured data; second, learn representations instead of handcrafting them; although neither principle is really new, they proved

*Corresponding author.
E-mail address: Gerard.Dreyfus@espci.fr (G. Dreyfus).

0304-3975/$ - see front matter
doi:10.1016/j.tcs.2005.08.026

very successful for handling structured data, to the point of generating a novel branch of numerical machine learning.

Keywords: Graph; Graph machine; Structured data; Recursive network; Folding network; Recurrent network; Neural network; SVM; QSAR; QSPR; RAAM; LRAAM; Classification; Regression

1. Introduction

The importance of learning from structured data was recognized in the earliest phases of the development of numerical machine learning. As early as 1988, it was mentioned in [15] that "graph recognition can be achieved (by neural networks); this may have an impact in various fields where graph recognition is important, such as picture processing, scene analysis, description of chemical structures, relational database systems, switching theory, etc.".

Not too surprisingly, the development of graph processing by machine learning paralleled the development of numerical machine learning. The very first attempts at handling structured data aimed at designing associative memories, either dynamic or static, for graphs or sequences. Then came the impressive development of numerical machine learning techniques (neural networks, graphical models, support vector machines) for regression and classification; the techniques and methods that evolved dealt essentially with non-structured data, an area of machine learning that is still extremely active and fruitful. It is only in recent years that handling structured data for regression and classification resurfaced as a major problem, spurred by potential applications, such as computer-aided drug design and image processing.

In the paper, the main approaches that have been considered thus far will be described; it will be shown that a basic principle, although not often stated explicitly, arises: *design structured machines for handling structured data*. The theorems that provide sound theoretical foundations to the approach will be outlined, and a small selection of applications will be described. Finally, open questions will be discussed.

2. Processing simple and complex sequences by dynamic learning associative memories

In his seminal paper [28], John Hopfield showed that an assembly of fully connected binary elements (McCulloch–Pitts neurons), with symmetrical connections and unit delays, behave as a discrete-time dynamical system whose dynamics exhibit fixed points. Such a system can act as an associative memory: at a given time, the state of a network of N neurons is described by a N-vector with binary components, which may be regarded as the binary code of some information. Under its free dynamics, the network evolves from an initial state to one of the fixed points; therefore, it associates the initial state, which may code for an incomplete or garbled piece of information, to a fixed point, which may code for the complete, noise-free piece of information. Therefore, it may "retrieve" information from an incomplete or partly erroneous version thereof.

In Section 2.1, the association between unstructured items (vectors) by dynamic learning machines (fully recurrent neural networks) will be described; from there, we will proceed to simple sequences, and to complex sequences. The processing of general graphs will be described in Section 3.

2.1. Processing vectors

The basic element of Hopfield networks, and variants thereof, is the McCulloch–Pitts neuron, a simple binary switching element that is governed (in its discrete-time version) by

$$y(k+1) = \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x}(k) - \theta), \tag{1}$$

where $\mathbf{x}(k)$ is the vector of variables at discrete time k (k positive integer), $\mathbf{w}$ is a vector of parameters ("weights") and θ is a threshold.

In the original Hopfield network, all neurons are mutually connected with symmetrical connections, so that a network of N such neurons obeys the following state equation

$$\mathbf{x}(k+1) = \operatorname{sgn}(\mathbf{W}\mathbf{x}(k) - \boldsymbol{\Theta}), \tag{2}$$

where $\mathbf{x}(k)$ is the state vector of the network at time k, $\mathbf{W}$ is the symmetric (N, N) matrix of the parameters of the network, and $\boldsymbol{\Theta}$ is the N-vector of thresholds; unless otherwise specified, the thresholds will be taken equal to zero in the following.

The ith fixed point of the dynamics of the network is defined as:

$$\mathbf{x}_i(k) = \mathbf{W}\mathbf{x}_i(k). \tag{3}$$

Therefore, the training of such a machine consists in finding the matrix $\mathbf{W}$ that guarantees the existence of n fixed points, coding for n pieces of binary information (vectors of dimension N) to be stored, defined by relation (3).

In the original form of the Hopfield model, a biology-inspired training algorithm—Hebb's rule—was applied. Hebb's rule states that the synaptic strength of a connection between two neurons, as modeled here by an element of matrix $\mathbf{W}$, increases if both neurons are active simultaneously. That can be roughly modeled by

$$\mathbf{W} = \frac{1}{N}\mathbf{X}\mathbf{X}', \tag{4}$$

where $\mathbf{X}$ is the (N, n) matrix whose columns are the components of the desired fixed points $\mathbf{x}_i$, and $\mathbf{X}'$ denotes the transpose of matrix $\mathbf{X}$. The application of that rule results in an approximate storage of the fixed points, provided that the fixed points are approximately orthogonal.

Taking a machine-learning oriented approach, without reference to biology, it was shown in [45] that any set of $n < N/2$ states can be made *exactly* the fixed points of the dynamics of the network, if training is performed by the *projection rule*, i.e. by computing matrix $\mathbf{W}$ as

$$\mathbf{W} = \mathbf{X}\mathbf{X}^{+}, \tag{5}$$

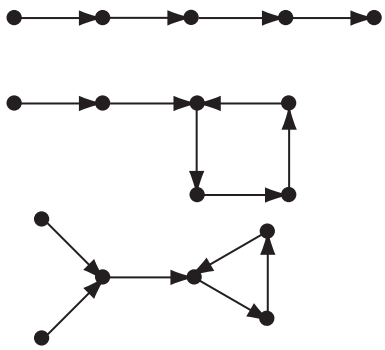


Fig. 1. Unary graphs; the final state of the upper graph is a stable state, while the final states of the two bottom graphs are cycles.

where $\mathbf{X}^+$ is the pseudo-inverse of $\mathbf{X}$. If the vectors of the desired fixed points are linearly independent, the pseudo-inverse matrix is given by

$$\mathbf{X}^+ = \mathbf{X}(\mathbf{X}'\mathbf{X})\mathbf{X}' \tag{6}$$

where $\mathbf{X}'$ denotes the transpose of matrix $\mathbf{X}$.

If the stored fixed points are orthogonal, the projection rule reduces to relation (4), i.e. to Hebb's rule.

In all the above cases, the dynamics of the machine is driven by a Lyapunov (or energy) function, which always decreases during the free evolution of the network:

$$L(k) = -\tfrac{1}{2}\mathbf{x}^{\mathbf{T}}(k)\mathbf{W}\mathbf{x}(k). \tag{7}$$

Therefore, the fixed points are the minima of that energy function. Other local minima ("spurious" or "garbage" fixed points) exist.

The associative memories thus described can also be used for classification purposes: some components of vector $\mathbf{x}$ are not used for encoding the information to be stored, but they code for a label [22]. For instance, a network with $N = 266$ is designed, where 256 bits are used to encode the picture of a handwritten digit, and 10 bits are used as a 1-out-of-10 code. Examples of each class of numerals (including the label) are stored as fixed points as described above. When an unknown digit must be recognized, the network is forced into its picture, with the code left blank; the network is subsequently allowed to evolve freely to one of the stored fixed points, whose label field indicates the class of the unknown pattern.

2.2. Processing simple sequences and cycles

Sequences can naturally be represented by graphs; each node of the graph is the state of the network at a given time, and each edge is a transition between two successive states. By simple sequence, we mean a sequence that has the structure of a unary graph, where each node has a single outgoing edge. A unary graph may be cyclic, as shown in Fig. 1—see also Fig. 5 of [46].

The task that is considered here is the storage and retrieval of sequences represented by unary graphs, which are stored as attractors of the dynamics of the network: when forced initially into a state (node of the graph) that is a binary code of a piece of information, the system should retrieve one of the nodes of the graph, thereby retrieving the graph itself since all transitions (edges of the graph) are stored in the network. Thus, the machine associates one item of information (e.g. a line of a poem) to a sequence (e.g. the full poem), and retrieves the latter even though the initial information may be garbled to a large extent.

The dynamics of the network obeys relation (2) as in the previous case, but the training algorithm is different. Instead of considering the storage of a set of n fixed points defined by (3), n transitions (the edges of the graphs) are stored: the matrix of parameters $\mathbf{W}$ is sought, such that, for each transition i from a given state $\mathbf{x}_i$ to a given state $\mathbf{y}_i$, one has:

$$\mathbf{y}_i = \mathrm{sgn}(\mathbf{W}\mathbf{x}_i). \tag{8}$$

Then matrix $\mathbf{W}$ is computed as

$$\mathbf{W} = \mathbf{Y}\mathbf{X}^+, \tag{9}$$

where $\mathbf{Y}$ is the (N, n) matrix whose columns are the target states $\mathbf{y}_i$, and $\mathbf{X}^+$ is the pseudo-inverse of matrix $\mathbf{X}$, whose columns are the states $\mathbf{x}_i$. By contrast to relation (5), matrix $\mathbf{W}$ is non-symmetric, a necessary condition for the storage of sequences.

A further improvement consists in using high-order neural networks, which take into account high-order correlations between simultaneous states of activities of the neurons [7,47]. In a Hopfield-type network as described above, the dynamics of neuron m is described by relation (2):

$$x_m(k+1) = \mathrm{sgn}\left(\sum_{j=1}^{N} w_{mj} x_j(k)\right).$$

In a high-order neural network, the latter relation becomes

$$x_m(k+1) = \mathrm{sgn}\left(\sum_{j=1}^{N} w_{mj} x_j(k) + \sum_{j=1}^{N}\sum_{l=1}^{j-1} w_{m,jl} x_j(k) x_l(k)\right) \tag{10}$$

thereby taking into account the correlation between the states of neurons j and l in the dynamics of neuron m. In a neural network of order 2, the parameters can be written into a $(N, N(N+1)/2)$ matrix $\mathbf{W}$, and a vector $\mathbf{z}$ can be defined, whose components are the x_m ($m = 1$ to N) and the $N(N-1)/2$ products $x_j x_l$ ($1 \leqslant j < l \leqslant N$). Then a transition from a state $\mathbf{z}_i$ to a state described by vector $\mathbf{x}_i$ can be written as:

$$\mathbf{x}_i = \mathrm{sgn}(\mathbf{W}\mathbf{z}_i). \tag{11}$$

Training consists in computing matrix $\mathbf{W}$ such that a set of n such relations are satisfied. Matrix $\mathbf{W}$ can be computed as

$$\mathbf{W} = \mathbf{X}\mathbf{Z}^+ \tag{12}$$

where $\mathbf{X}$ is the (N, n) matrix whose columns are the vectors $\mathbf{x}_i$, and $\mathbf{Z}^+$ is the pseudo-inverse of the $(N(N+1)/2, n)$ matrix $\mathbf{Z}$ whose columns are the vectors $\mathbf{z}_i$.

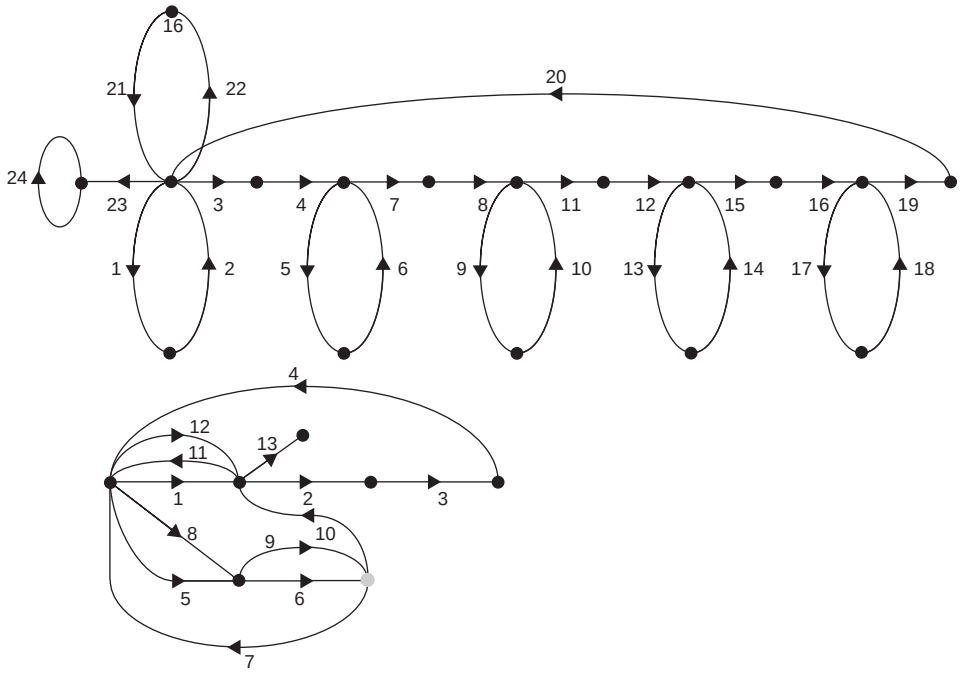


Fig. 2. Two complex sequences (from [21]). Transitions are numbered from 1 to the total length of the sequence. Top sequence: $v = 2$; bottom sequence: $v = 4$.

As in the case of the storage of fixed points of the dynamics, a labeling scheme can be implemented, as described in Section 2.1.

2.3. Processing complex sequences

By complex sequences, we mean sequences that, in contrast to simple sequences, have the structure of non-unary graphs: a given node may have several outgoing edges. Examples of complex sequences (from [21]) are shown in Fig. 2. In order to process such sequences, the state of the network at time $k+1$ must not depend on the state of the network at time k only, but also on the state at times $k-1, k-2, \ldots, k-v+1$. Therefore, the dynamical system is described by Nv first-order [1] recurrent equations instead of N equations for storing simple sequences. For the top graph of Fig. 2, one has $v = 2$, while, for the bottom graph, one has $v = 4$: when the system is in the state shown as a gray node, its next state depends not only on its present state, but also on the three previous ones.

[1] Here, the term "order" means the number of recurrent variables of a recurrent equation, as usual for discrete-time dynamical systems; it is not to be confused with the "order" of a "high-order neural network" as defined in Section 2.2, where it is the degree of the polynomial computed by the neuron prior to thresholding.

The processing of such sequences can be performed in several ways; the simplest one, in the spirit of the previous section, consists in defining the dynamics of a neuron m by:

$$x_m(k+1) = \operatorname{sgn}\left(\sum_{j=1}^{N} w_{mj}^{1} x_j(k) + \sum_{j=1}^{N} w_{mj}^{2} x_j(k-1) + \cdots + \sum_{j=1}^{N} w_{mj}^{v} x_j(k-v+1)\right). \tag{13}$$

The parameters of the machine can be written into a (N, Nv) matrix, and a vector $\mathbf{z}$ can be defined as the concatenation of vectors $\mathbf{x}(k)$, $\mathbf{x}(k-1), \ldots, \mathbf{x}(k-v+1)$. Then a transition from a state $\mathbf{x}_i$ to a state described by vector $\mathbf{z}_i$ can be written as:

$$\mathbf{x}_i = \operatorname{sgn}(\mathbf{W}\mathbf{z}_i). \tag{14}$$

As in the previous sections, training consists in computing the matrix $\mathbf{W}$ such that a set of n such relations are satisfied. Matrix $\mathbf{W}$ can be computed as

$$\mathbf{W} = \mathbf{Z}\mathbf{X}^{+}, \tag{15}$$

where $\mathbf{X}$ is the (N, n) matrix whose columns are the vectors $\mathbf{x}_i$, and $\mathbf{Z}^{+}$ is the pseudo-inverse of the (Nv, n) matrix $\mathbf{Z}$ whose columns are the vectors $\mathbf{z}_i$.

Interestingly, the training of sequences—whether simple or complex—can be performed by other machine learning algorithms, such as the Perceptron learning rule [21]. A Perceptron [51] is a learning machine that is intended to perform 2-class classification. An item j to be classified, described by a vector of features $\mathbf{z}_j$, is assigned a label $x_j = +1$ if it belongs to one of the classes, say class A, and $x_j = -1$ if it belongs to class B. The Perceptron computes the quantity

$$\operatorname{sgn}(\mathbf{w} \cdot \mathbf{z}_j), \tag{16}$$

where $\mathbf{w}$ is the vector of parameters of the Perceptron; item j is correctly classified if the latter quantity is equal to x_j, i.e. iff:

$$x_j(\mathbf{w} \cdot \mathbf{z}_j) > 0. \tag{17}$$

The number of such inequalities is equal to the number of examples. Thus, the problem of Perceptron learning is that of finding a vector $\mathbf{w}$ that satisfies the set of inequalities (17). If the examples are linearly separable, then the Perceptron algorithm is guaranteed to solve the problem in finite computation time [51].

Reverting to the learning of sequences, relation (14) can be rewritten as:

$$\mathbf{x}_i \cdot \mathbf{W}\mathbf{z}_i > 0. \tag{18}$$

Therefore, the problem of storing n sequences in a network of N neurons is equivalent to finding N Perceptrons (one Perceptron per neuron), that correctly classify n examples, each of which being described by a vector of dimension $d = N$ (in the case of relation (8)), $d = N(N+1)/2$ (in the case of relation (11)), or $d = Nv$ (in the case of relation (14)). Insofar as the number of sequences n is smaller than $d/2$, linear separability of the

examples is guaranteed by Cover's theorem [9], so that a solution to (18) can be found by the Perceptron algorithm.

In all the above cases, the machine was a dynamic system that was left to evolve freely, i.e. without any external input. Such inputs can be reinstated in those machines by making use of the thresholds that were mentioned in relation (2), which can be viewed as nonzero, constant, external inputs. The idea of using the Perceptron algorithm, or the more powerful Support Vector Machine algorithm, for designing a machine that is guaranteed to perform a set of binary sequences in response to a given set of constant external inputs was developed recently [50] for biological modeling purposes.

3. Processing general graphs by dynamic learning machines

The earliest attempts at processing graphs, other than sequences, with learning machines, were motivated by the modeling of the operation of the visual system [57,4]. The main assumption underlying this approach is that the information might be encoded in the brain as a connectivity graph, rather than as patterns of neuronal activity. As a consequence, the dynamics of the system is governed by a Lyapunov function that, as opposed to relation (7) for instance, is a function of the time-varying parameter matrix $\mathbf{W}(k)$ rather than a function of the neuronal states $\mathbf{x}(k)$. Therefore, the associative memory is designed for storing and retrieving patterns of connectivity rather than patterns of neuronal activity. The description of the details of that theory lies beyond the scope of the present paper. Interestingly however, it gave rise to the "elastic matching" technique [3] for shift- and deformation-invariant pattern recognition.

An alternative approach was considered in [35,15], which is more closely related to the learning machines described in Section 2. The task is to associate automatically an unknown graph to one element of a set of predefined graphs; therefore, it is again an associative memory task, as described above, except for the fact that the items to be associated are general graphs, instead of being vectors or sequences. The machines feature two elements:

- a preprocessor, which generates graphs that are isomorphic to the unknown graph, for instance by performing two-vertex exchanges,
- an associative memory as described in Section 2.1, which codes for the graph and retrieves the stored graph that is most similar to the graph resulting from the transformation of the original one by the preprocessor.

The principle of operation takes advantage of the property, mentioned in Section 2.1, that the dynamics of the associative memory is governed by a Lyapunov function.

Consider a graph with at most N nodes, and its (N, N) adjacency matrix, where $+1$ codes for the presence of an edge and -1 for the absence of an edge. That matrix can be encoded into a Hopfield network having $N(N-1)/2$ neurons, which can be trained as explained above in order to store the graphs that should be retrieved. An unknown graph is also encoded by its adjacency matrix $\mathbf{G}$. The graph transformation performed by the preprocessor can be encoded as a permutation matrix $\mathbf{T}$, so that the adjacency matrix of the transformed graph is $\mathbf{T}'\mathbf{G}\mathbf{T}$, which can be encoded in a similar way by the associative memory network. Therefore, the whole task can be depicted as an optimization task. In [35], the optimization is performed by the simultaneous operation of a Hopfield–Tank optimizer

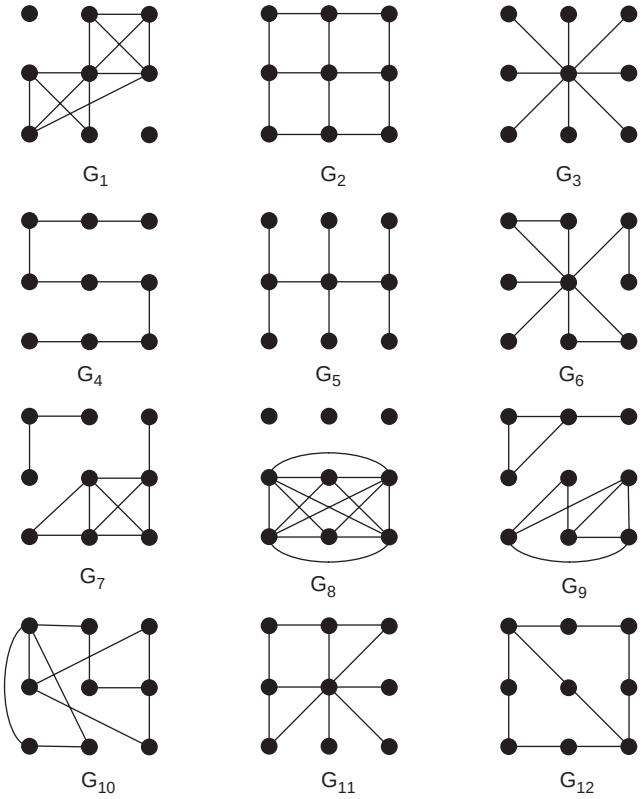


Fig. 3. Twelve graphs stored in the memory network.

[29] and a Hopfield associative memory; the description of the Hopfield–Tank optimizer network is beyond the scope of the present paper. In [15], the optimization of the "energy" of the associative memory is performed by simulated annealing. If the unknown graph is isomorphic to one of the stored graphs, the latter is retrieved by simulated annealing alone since the stored graphs are minima of the memory networks; if the unknown graph is not isomorphic to one of the stored graphs, then the memory network is forced into a state that codes for the graph found by simulated annealing, and it is left to evolve under its own dynamics. Thus, either one of the stored graphs is retrieved, or a spurious fixed point is found; in the latter case, the unknown network is not recognized as similar to one of the stored ones.

Fig. 3 shows 12 graphs stored in a memory network, and Fig. 4 shows how one of the stored graphs is retrieved if the unknown graph is isomorphic to one of the stored graphs, but with two missing edges.

A generalization of Hopfield networks, embodying the structure of recursive auto-associative memories (RAAMs), will be described in Section 4.

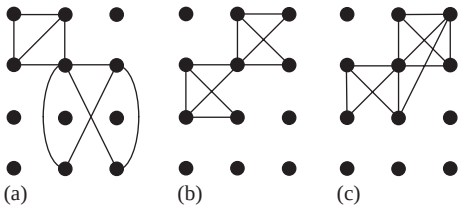


Fig. 4. (a) Unknown graph, isomorphic to graph G_1 with two edges missing; (b) graph found after simulated annealing; (c) graph G_1 is retrieved by the memory network.

4. Processing graphs by static associative memories: RAAMs and their variants

The first attempts at handling structured data (sequences and graphs) by numerical learning machines were performed with *dynamic* associative memories, as described in the previous sections. We now turn to algorithms developed later for handling structured data, by making use of *static* associative memories: RAAMs first described by Pollack in [48].

A RAAM is essentially a feedforward neural network. Therefore, we first recall basic concepts and definitions pertaining to feedforward neural networks.

4.1. Feedforward neural networks: basic concepts and definitions

In its simplest setting, a feedforward neural network (also termed "Multilayer Perceptron" or MLP) is a weighted sum of parameterized nonlinear functions called "hidden neurons" (see for instance [5,14]); its output is given by

$$g(\mathbf{x}) = \sum_{i=1}^{N_c} w_{N_c+1,i} f_i(\mathbf{x}) = \mathbf{w}_1 \cdot \mathbf{f}(\mathbf{x}), \tag{19}$$

where $g(\cdot)$ is the output value of the neural network, $\mathbf{f}(\mathbf{x})$ is the N_c-vector of the outputs of the hidden neurons, $w_{N_c+1,i}$ is the weight of hidden neuron i in the weighted sum that is the output value of the network, and $\mathbf{x}$ is the N-vector of the variables ("inputs") of the network; $\mathbf{w}_1$ is the N_c-vector of the parameters ($w_{N_c+1,i}$, $i = 1$ to N_c). Therefore, the network performs a mapping of $\mathbb{R}^N$ to $\mathbb{R}$. A pictorial representation of that neural network is shown in Fig. 5.

In the following, unless otherwise specified, a "neuron" is a "soft" version of the McCulloch–Pitts neuron defined by relation (1)

$$f_i(\mathbf{x}) = \tanh \sum_{j=1}^{N} w_{ij} x_j = \tanh(\mathbf{w}^i \cdot \mathbf{x}), \tag{20}$$

where $\mathbf{w}^i$ is the N-vector of the parameters pertaining to neuron i. Any s-shaped ("sigmoid") function other than the tanh function can be used as well. We denote by $\mathbf{W}_2$ the (N_c, N) matrix whose rows are the vectors $\mathbf{w}^i$, $i = 1, \ldots, N_c$.

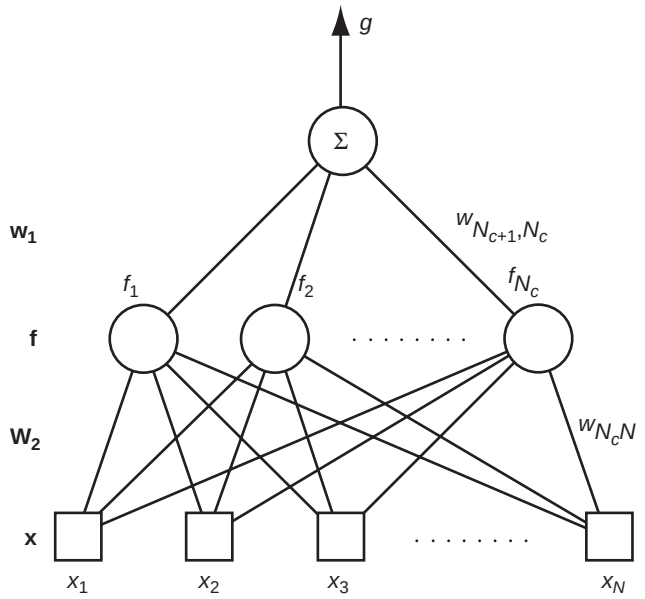


Fig. 5. The simplest feedforward neural network. Void circles stand for neurons as described by relation (20) for instance. The circle enclosing Σ sign performs a weighted sum. Squares do not perform any computation.

Therefore, the equation of the simplest feedforward neural network is [2]:

$$g(\mathbf{x}) = \sum_{i=1}^{N_c} w_{N_c+1,i} \tanh\left(\sum_{j=1}^{N} w_{ij} x_j\right). \tag{21}$$

Thus:

$$g(\mathbf{x}) = \mathbf{w}_1 \cdot \mathbf{f}(\mathbf{W}_2 \mathbf{x}). \tag{22}$$

Feedforward neural networks can approximate any sufficiently regular function with arbitrary accuracy, provided the number of hidden neurons is appropriate [30,31]. In practice, feedforward neural networks are not used for function approximation, but for data modeling: given a finite set of input–output pairs ($\{\mathbf{x}_k, y_k\}$, $k = 1, \ldots, n$), called "training set", a model is sought, which provides the best predictions of data that is not present in the training set. In most cases, training consists in estimating the parameters of the network by minimizing the least squares cost function, i.e. the sum of the squared modeling errors

$$J(\mathbf{w}) = \sum_{k=1}^{n} (y_k - g(\mathbf{x}_k))^2, \tag{23}$$

[2] For simplicity, a constant input (called "bias" of the network) to the hidden and output neurons has been omitted in the equations and figures. It is nevertheless very important for all practical purposes.

where $\mathbf{w}$ is the vector of all parameters of the network. Since g is not linear with respect to the parameters of matrix $\mathbf{W}_2$, the cost function (23) is not quadratic with respect to those parameters. Therefore, the usual least squares technique is not appropriate for estimating the parameters of the model: one has to resort to nonlinear optimization techniques, which update iteratively the parameters of the model until a minimum of the cost function is found. Those optimization techniques make use of the value of the gradient of the cost function with respect to the parameters. The gradient of the cost function can be computed exactly in a number of ways, the most computationally efficient of which being known as *backpropagation*. Several gradient-based optimization methods exist; the simplest—and by far the most inefficient—method, is simple gradient descent, whereby the parameters are updated iteratively in proportion to the current value of the gradient of the cost function. Second-order, Newton-like techniques are much more efficient, slashing computation times by orders of magnitude. Note that, in the early neural network literature, backpropagation meant both the computation of the gradient of the cost function and the optimization of the latter by simple gradient descent: early "connectionists" were not aware of the existence of second-order optimization methods, although the latter were widely used, in signal processing for instance.

Also note that backpropagation is definitely not the only way of computing the gradient of the cost function with respect to the parameters. Various other methods were developed for the adaptation of recursive filters before neural networks came to existence (recurrent neural networks *are* nonlinear recursive filters). For a discussion of backpropagation versus adaptive filtering algorithms, see [40].

Several straightforward extensions of the simple feedforward neural network exist. First, as will frequently be the case in the present paper, neural networks may have several outputs instead of just one, computing different functions of the outputs of the same hidden neurons. Then the network computes an N_g-vector $\mathbf{g}(\mathbf{x})$ whose mth component is

$$g_m(\mathbf{x}) = \sum_{i=1}^{N_c} w_{N_c+m,i} \tanh\left(\sum_{j=1}^{N} w_{ij} x_j\right) \tag{24}$$

and the cost function for training becomes:

$$J(\mathbf{w}) = \sum_{m=1}^{N_g} \sum_{k=1}^{n} (y_k - g_m(\mathbf{x}_k))^2. \tag{25}$$

Thus, the network performs a mapping of $\mathbb{R}^N$ to $\mathbb{R}^{N_g}$. The training set is a collection of input–output pairs $(\{\mathbf{x}_k, \mathbf{y}_k\}, k = 1, \ldots, n)$, where $\mathbf{x}_k$ is the N-vector of variables pertaining to example k, and $\mathbf{y}_k$ is the N_g-vector of outputs pertaining to example k.

As a further extension, the output(s) may be computed in the same way as the hidden neurons, i.e. as nonlinear function(s) of a weighted sum of the outputs of the hidden neurons. If the nonlinear function is a tanh, then the output(s) of the neural network is (are) constrained to lie in $[-1, +1]$:

$$g_m(\mathbf{x}) = \tanh\left[\sum_{i=1}^{N_c} w_{N_c+m,i} \tanh\left(\sum_{j=1}^{N} w_{ij} x_j\right)\right]. \tag{26}$$

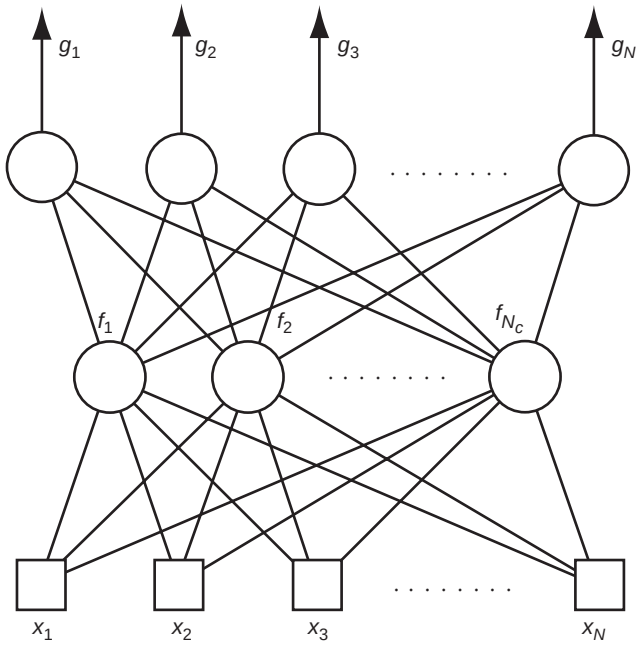


Fig. 6. Encoder–decoder network. Void circles stand for neurons as described by relation (20). Squares do not perform any computation.

Such is usually the case when neural nets are intended to perform classification tasks. It is also the case for RAAMs as described in the next section.

4.2. Recursive auto-associative memories

RAAMs [48] are intended to provide a compact representation of trees. The basic element of the RAAM is essentially an encoder–decoder feedforward neural network with $N_g = N$ and $N_c < N$: the number of outputs is equal to the number of variables, and the number of hidden neurons is smaller than the number of variables, as shown in Fig. 6. In addition, the output neurons are tanh functions of a weighted sum of the outputs of the hidden neurons, as in relation (26).

The network is trained to perform auto-association, i.e. the input–output pairs of the training set are of the form $(\{\mathbf{x}_k, \mathbf{x}_k\}, k = 1, \ldots, n)$. If the problem has a solution, then, given a binary input vector, the outputs of the hidden neurons may be viewed as a compact code of the inputs, from which the input may be reconstructed. After successful training, feeding the network with an input vector $\mathbf{x}$ generates a vector of real numbers $\mathbf{f}(\mathbf{x})$ at the output of the hidden neurons; therefore, the layer of hidden neurons acts as an encoder; conversely, if the hidden neurons are forced into that state, the original information will be retrieved at the output of the network, thereby acting as a decoder.

In the framework of graph processing, RAAMs have the ability of generating codes for trees. Consider, as in [48], the binary tree $((AB)(CD))$, as shown in Fig. 7, where

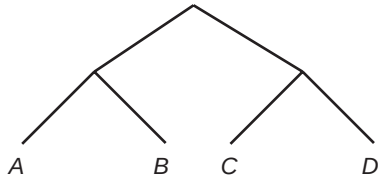


Fig. 7. A binary tree.

A, B, C, D are symbols that are coded on b bits, e.g. in a 1-out-of-b code. It is desired to design a learning machine, involving one or more encoder–decoder networks as described above, which can find, by training, a code for such binary trees. In [48], the author introduces an ad hoc dynamics *of training*: the encoder–decoder pair should first be trained to auto-associate (AB) to (AB), resulting in a code C_1, then (CD) to (CD), resulting in a code C_2, and then trained to auto-associate (C_1C_2) to (C_1C_2), resulting in a code C_3. Therefore, a part of the training set (C_1 and C_2) changes during training (the so-called "moving target" problem), an awkward situation in which the very convergence of the training procedure is not guaranteed.

We propose here a different approach, which does not involve any specific dynamics of training, and, in addition, raises a general issue: *the isomorphism of the structure of the machine to be trained with the graph to be encoded*. Remember that we are looking for a machine that can, with the same set of parameters, find a code C_1 for (AB), a code C_2 for (CD) and a code C_3 for (C_1C_2). That is performed by the machine shown in Fig. 8, a feedforward neural network with *shared weights* [58]: the "connections" are not drawn in detail as in Fig. 6, but they are shown as gray areas; areas that have the same shade of gray depict sets of "connections" that have the same vector of parameters.

Note that, in that simple setting, the operation of the RAAM for graph encoding requires that the size of the hidden layer be equal to the number of bits necessary for encoding a leaf of the tree. This is unfortunate because there is no reason why the complexity of the input–output mapping, which is reflected in the number of hidden units, should have any relation to the number of neurons necessary for encoding or decoding the leaves. This problem may be circumvented by using more than one hidden layer in the encoder and/or in the decoder parts, thereby allowing for a greater complexity of the mapping. Therefore, in the following figures, the shaded areas may represent not just a single layer of parameters, but two layers of parameters and a layer of hidden neurons; then areas of the same shade of gray represent identical weights and hidden neurons.

Interestingly, the structure of the network reflects the structure of the tree that it is intended to encode, as shown in Fig. 8. We will see later that this is a general concept: designing a learning machine whose structure is similar to that of the graph from which it is intended to learn.

This approach solves the "moving target" problem; however, it does not solve the termination problem that standard RAAMs have: when a code is input to the decoder, there is no way to know whether the result is another code, or leaves of the tree. The fact that the codes are real numbers and the leaves are binary may not be sufficient if training is not perfect; see [4,8] for a discussion of that problem.

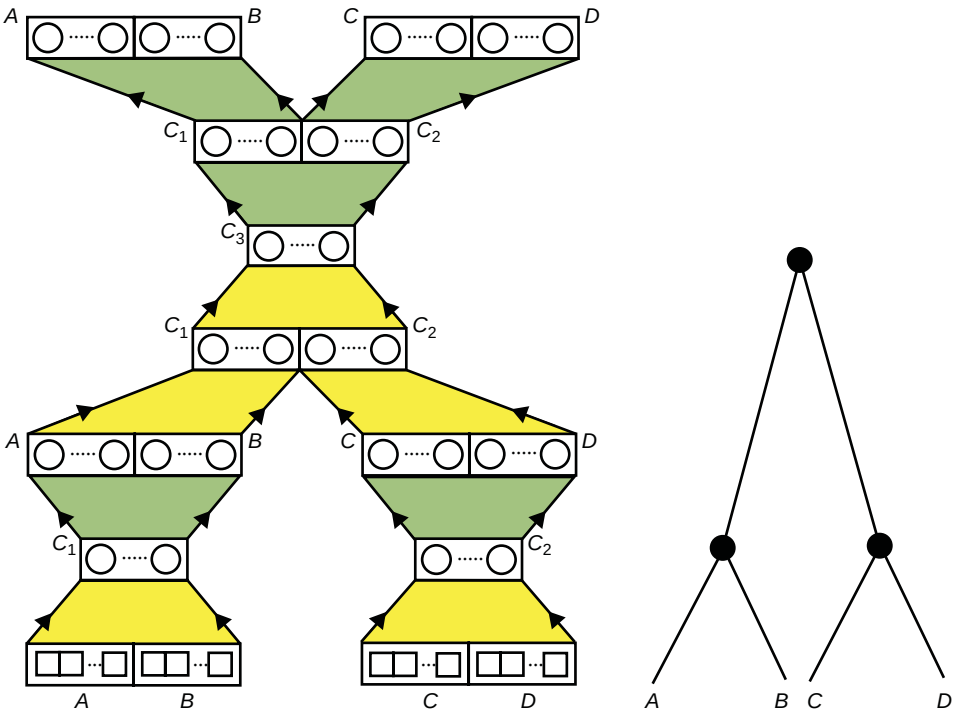


Fig. 8. Encoding the binary tree shown in Fig. 7. Left: structure of the network to be trained; void circles stand for neurons as described by relation (20); squares do not perform any computation; the topmost layer of connections is necessary for training only; areas with the same shade of gray depict identical vectors of parameters (shared weights). Right: graph structure of the network; black dots denote encoder–decoder pairs.

Note also that the network might be viewed as a recurrent neural network (Fig. 9), whose equation is

$$\mathbf{g}(k) = \Psi(\mathbf{g}(k-1)), \tag{27}$$

where $\mathbf{g}(k)$ denotes the output vector of the network at discrete time k, and Ψ denotes the function computed by the feedforward network shown in Fig. 8. If the latter network has been trained successfully, then the stored graphs are fixed points of the dynamics of the recurrent network. Therefore, the network can be used for graph retrieval, in the spirit of the Hopfield networks discussed in Section 2, albeit with a completely different training algorithm.

A RAAM can encode sequences, which may be viewed as special instances of binary graphs. Fig. 10 shows the network to be trained. Just as in the previous case, the structure of that network reflects the structure of the tree for which a representation is sought.

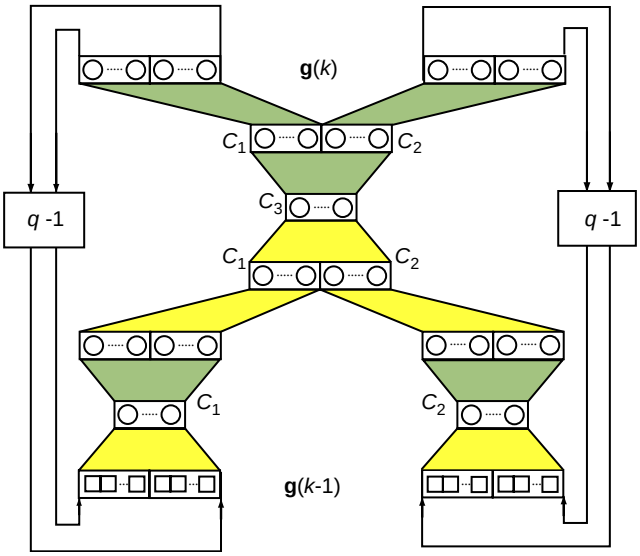


Fig. 9. The network of Fig. 8 can be viewed as a recurrent network; training results in storing the desired trees as fixed points of the dynamics of the network; q^{-1} is the usual backward shift operator.

4.3. Labeling RAAMs

RAAMs are restricted to encoding trees; in order to handle labeled graphs, RAAMs can be modified as follows [55]: each node of the graph is described by a unique label, which is a vector of binary variables, and by a set of pointers to its children in the graph, which are vectors of real numbers; all nodes have the same number of pointers, which is equal to the valence of the graph (the maximum number of outgoing edges). All pointers are encoded by the same number of neurons, hence are vectors of the same size. For each node, an encoder is fed with the set of variables that code for the label of the node and for the pointers to its children in the graph; the hidden units encode the pointer to the node. Thus, the vector of the outputs of the decoder contains real numbers (the pointers) that are obtained by training. Therefore, the training set is dynamic insofar as it involves elements that are computed during training. A similar situation ("moving target") was described in Section 4.2 for RAAMs. That difficulty can be circumvented, as described in that section, by the *shared weights* technique.

As an illustration, consider, as in [55], the labeled graph shown in Fig. 11. The codes for the pointers can be learnt by training the network shown in Fig. 12, where $\mathbf{L}_i$ stands for the label of node i, and $\mathbf{p}_i$ stands for the real-valued vector of the pointer (to be learnt) to node i. The fields of void pointers are denoted by *nil*. Again, as in examples shown in the previous section, the structure of the network reflects the structure of the graph to be encoded.

Just as for RAAMs, the network may be viewed as a recurrent one, and its training may be viewed as training the encoded tree as a fixed point of the dynamics of the network. Therefore, RAAMs can be used as content-addressable memories, similarly to Hopfield networks [55].

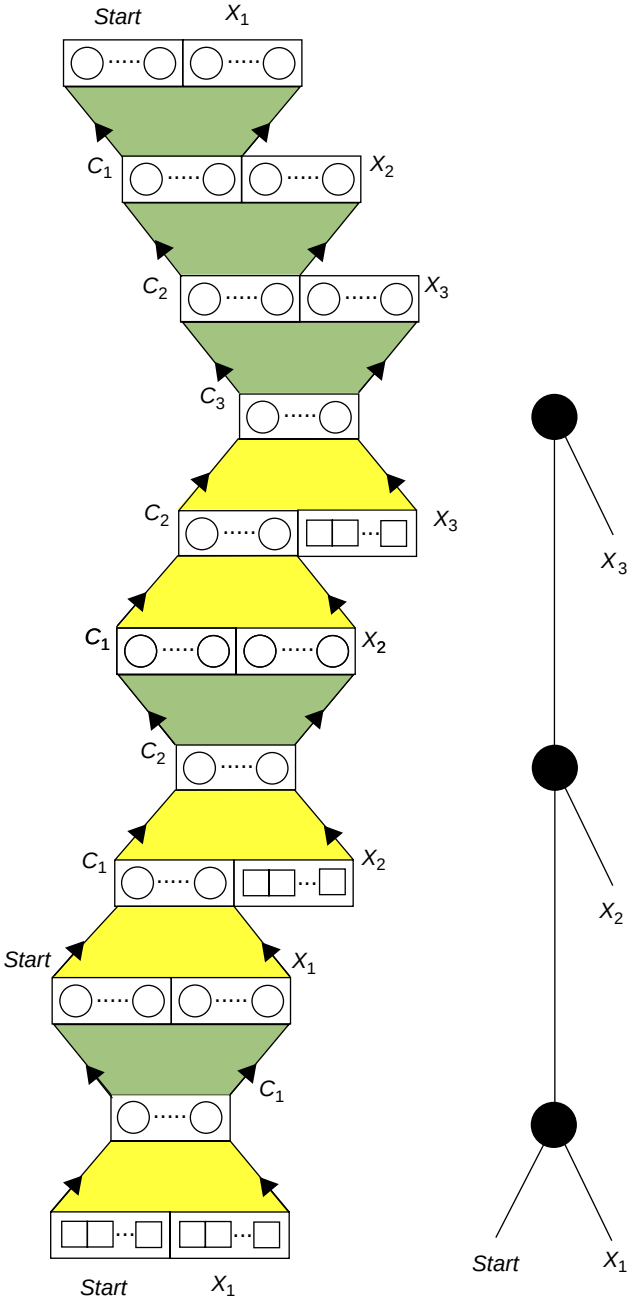


Fig. 10. Left: structure of the network to be trained; void circles stand for neurons as described by relation (20); squares do not perform any computation; the topmost two layers of connections are necessary for training only. Right: graph structure of the network; black dots denote encoder–decoder pairs.

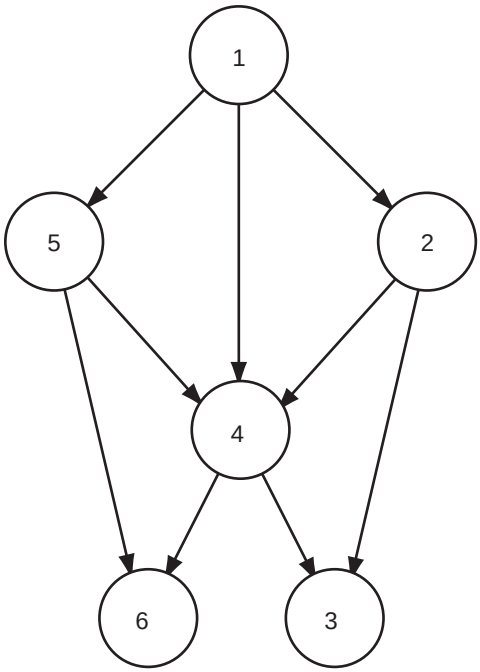


Fig. 11.

LRAAMs were initially designed for encoding acyclic graphs. We show here that cyclic graphs can be encoded too. Consider the cyclic graph shown in Fig. 13; for the cycle $1 \to 2 \to 3 \to 1$ to be causal, let us insert a unit time delay in one of its edges, say edge $1 \to 2$. Then the graph can be encoded by the recurrent network shown in Fig. 13. The network is in the canonical form of a recurrent neural network [13], with state vector $\mathbf{p}_2$ and output vector $[\mathbf{L}_4\, \mathit{Nil}\, \mathit{Nil}]^{\mathrm{T}}$. The symbol *Nil* can be encoded in any appropriate way. If the necessary delay had been assigned to another edge of the cycle, a different, but equivalent, LRAAM would have been found. The network can be trained to produce sequences of constant outputs $[\mathbf{L}_4\, \mathit{Nil}\, \mathit{Nil}]^{\mathrm{T}}$ in response to sequences of constant inputs $[\mathbf{L}_4\, \mathit{Nil}\, \mathit{Nil}]^{\mathrm{T}}$. The length of the sequences must be chosen appropriately to accommodate the response time of the network; typically, it should be at least as large as the order of the network, i.e. as the number of state variables. The training of the recurrent network can be performed by well-established techniques, see for instance [43].

5. Graph regression and classification

The techniques for handling graphs that were described in the previous section aimed essentially at showing that "connectionist" approaches had the ability of handling symbolic data, an important question that will not be discussed here; for a thorough discussion of that issue, see for instance [6]. In the present section, we discuss a related but somewhat different

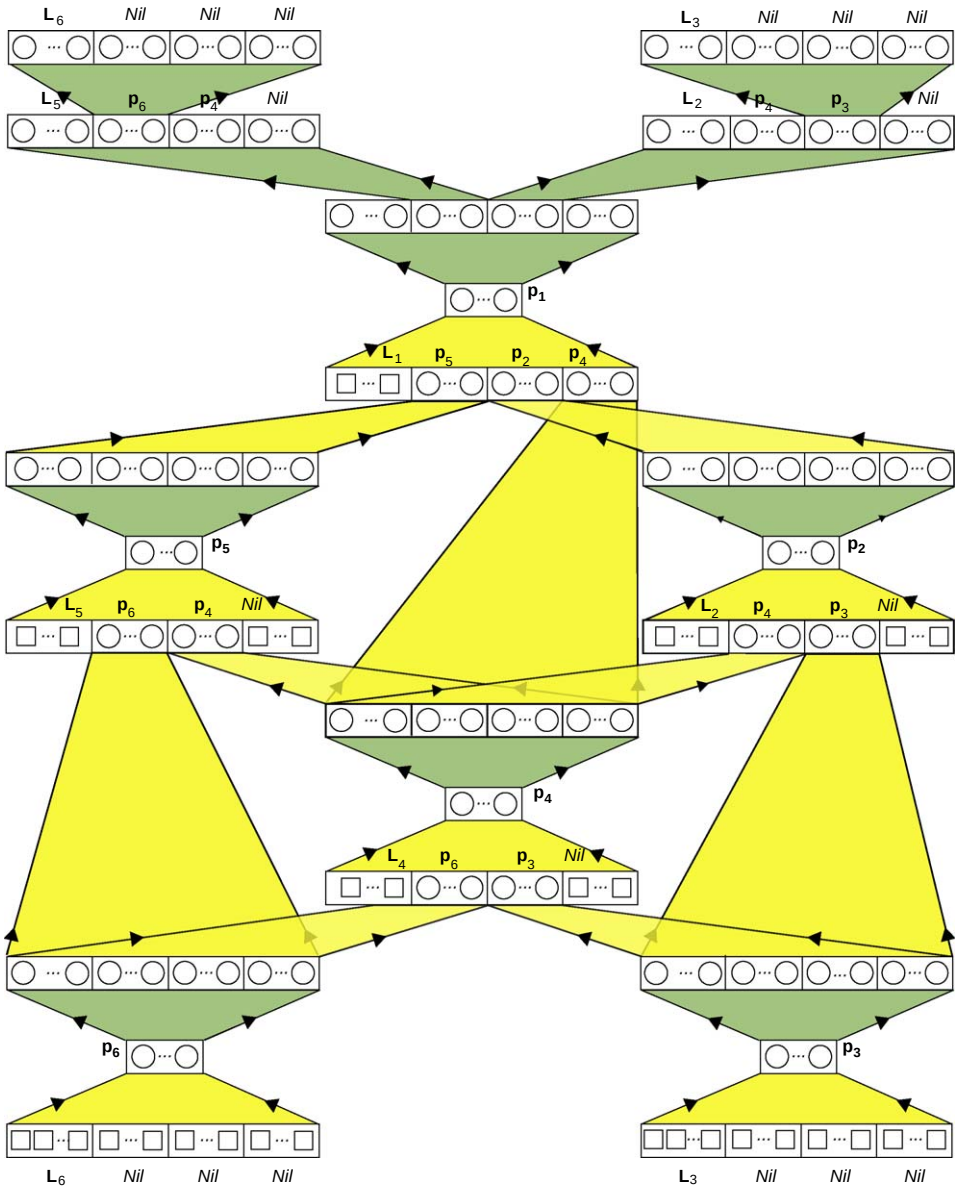


Fig. 12. Encoding the labeled graph of Fig. 11; void circles stand for neurons as described by relation (20); squares do not perform any computation; the topmost two layers of connections are necessary for training only.

problem, namely, the ability of mapping graphs to vectors, just as conventional numerical learning machines map vectors to vectors. In other words, we show that the usual tasks that are addressed in conventional numerical machine learning can also be addressed when structured information must be handled. We further show that, although the purpose is different,

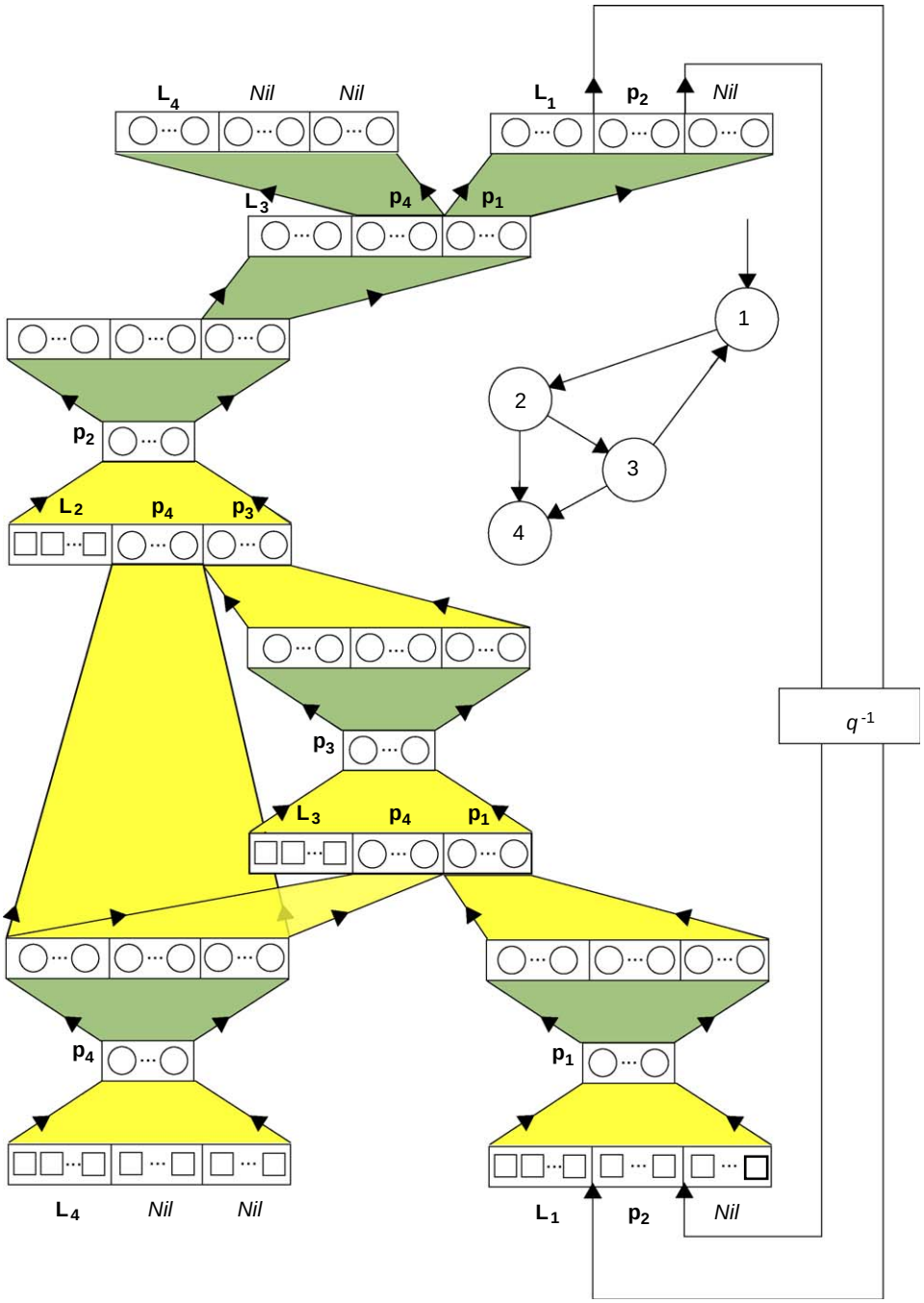


Fig. 13. Encoding the cyclic labeled graph shown, with a recurrent neural network.

the basic idea that is useful for these new developments was potentially present in the early approaches, although often not stated explicitly: the idea of *designing a learning machine whose structure reflects the structure of the graph from which information is to be learnt*.

5.1. Regression and classification with acyclic graphs

The first step toward that idea was taken in [18], where the purpose was a classification task on trees: the authors argued that, since an LRAAM was able to find a vector representation of a tree, that representation might be used as input to a classifier, e.g. a neural network. They further argued that it was not necessary to first learn the representation, and subsequently use it for classification: by training simultaneously the encoder and the classifier, a representation of the graph that might not be complete, but still be appropriate for the classification task at hand, might evolve. That is actually similar to the principle of "convolutional networks" [36], which learn simultaneously the classification task *and* the picture representation that is useful for that task, instead of handcrafting the representation prior to training a classifier. In [18], it was further argued that LRAAMs must be trained in the "moving target" context, whereas their classifier can be trained by "backpropagation through structure"; however, we have shown, in Sections 4.2 and 4.3, that RAAMs and LRAAMs can indeed be trained in a standard fashion provided the structure of the network is the same as the structure of the tree to be encoded (Fig. 8), and shared weights are used. Similarly, a classifier from tree-structured data can be designed by dropping the decoder part, and it can be trained by any standard method: computation of the gradient by backpropagation *with shared weights*, and minimization of the least squares cost function by any good gradient method (BFGS, Levenberg–Marquardt, conjugate gradient, see for instance [49]).

Clearly, classification is not the only task that can be performed by such a structure: regression can be performed as well, the sole difference being that the output is real-valued instead of being binary. Fig. 14 shows a machine built along those lines, intended to perform classification or regression on data having the tree structure displayed in Figs. 7 and 8. The lower part of the machine, using shared weights, can be computed recursively, hence the term "recursive network" used for such networks (also termed "folding networks"), but it is basically a feedforward neural network making use of shared weights, with a structure that reflects the structure of the data.

It may be useful, at this point, to draw a parallel between the present approach and the "dynamic semi-physical modeling" approach [44] to process modeling: in the latter case, the structure is not present in the data, but it is present in the prior knowledge of the process; therefore, a neural model is designed and trained, whose structure reflects the equations derived from the physical (or chemical, biological, etc.) knowledge. Thus, *whether structure is present in the data or in the process, structured learning machines should be designed, with a structure that reflects either the structure of the data or the structure of prior process knowledge*.

5.2. Regression and classification with cyclic graphs

Up to this point, we focused on the case of directed acyclic graphs. We now extend the approach to cyclic graphs. In Section 4.3, we showed that a simple cyclic graph can be

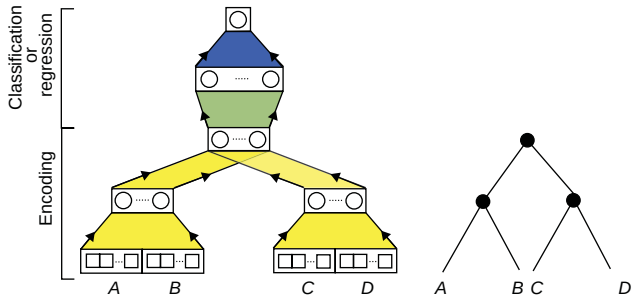


Fig. 14. A machine for performing regression or classification on data having the tree structure shown on the right; the bottom two layers of hidden neurons are the "encoder", while the third layer of hidden neurons and the output neuron perform the classification or regression task.

encoded by a LRAAM by introducing a "delay" in the appropriate edges of the graph cycles, and training the LRAAM as a dynamic (recurrent) neural network whose state variables are the components of the pointer vector of one of the nodes of the "delayed" edges; in the case of classification or regression, the state variables are the outputs of nodes related to the cycles. As an illustration, Fig. 15 shows a network that can perform classification or regression on the cyclic graph shown in Fig. 13. For complex graphs with many cycles, finding the combination of "delayed" edges that generates the simplest recurrent graph, i.e. the graph that has the smallest number of state variables (recurrent connections) may be an intricate task, even for relatively small graphs.

However, it was shown in [13] that any graph with delays can be cast into a canonical form, made of an acyclic graph and a minimal number of recurrent connections; it was further shown that an optimal solution can be found automatically in polynomial time. Fig. 16 shows an example of a cyclic graph with four cycles, and four delays, but actually only two state vectors ($2n_h$ state variables, where n_h is the number of hidden neurons). Other optimal encodings can be found for the same graph, but the number of state variables is invariant.

5.3. Regression and classification from arbitrary graphs: graph machines

In many applications, the problem at hand requires training from graphs that are a set of trees. Then, since a tree is a universal structure, a single neural network with a tree structure can be designed, with appropriate height and fan-out: the examples of the training set differ in their labels only.

If the problem at hand requires training from different kinds of graphs, say trees, non-tree acyclic graphs, and cyclic graphs, the problem becomes more complex. One has to train a set of machines that have the structure of the graphs from which training must be performed, with shared weights *within each machine*, as explained above for recursive networks, and *across the different machines*. Such a set of machines is termed *graph machine* [20]; the number of machines may be equal to the number of examples, and, instead of training a single machine with several examples, several machines are trained with one example each,

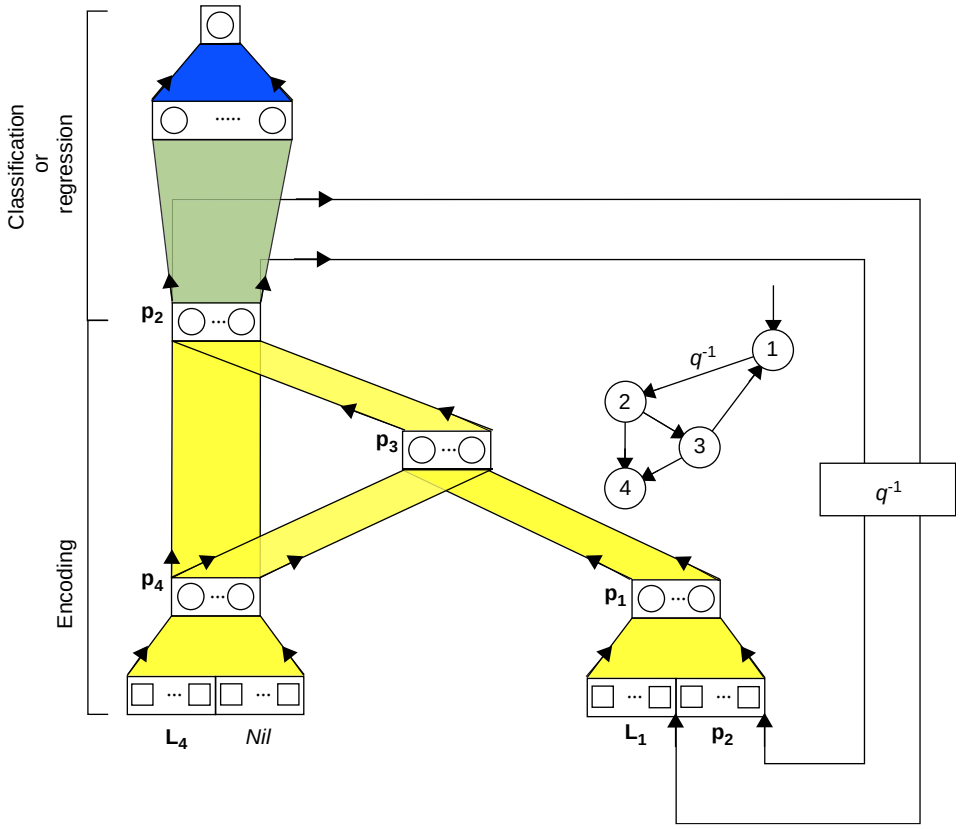


Fig. 15. A machine that can perform classification or regression with cyclic graphs. In the cycle $1 \to 2 \to 3$, a delay q^{-1} was assigned to edge $1 \to 2$. The notation $\mathbf{p}_i$ has been kept from the LRAAMs, although they are no longer pointers. *Nil* may be assigned any value, e.g. 0.

which is possible since weights are shared across the machines. In addition, provision is made for handling cyclic graphs, as will be described in Section 6.1.

5.4. Theoretical foundations of graph regression and classification

Conventional neural networks are universal approximators: any reasonable function can be approximated uniformly by a neural network with one layer of hidden neurons with sigmoidal nonlinearity and a linear output neuron [30,31]. The universal approximation property was shown to hold true for recurrent networks in [53,54]. Then a natural question arises: is there any equivalent solid result for the case of structured data? The machines described in the above sections have two parts: an encoding part that provides a code for the graph, and a classification or regression part, which is a standard machine whose input is the representation of the graph. Since the universal approximation property applies to the latter part, the open question is that of the representational capability of the encoding

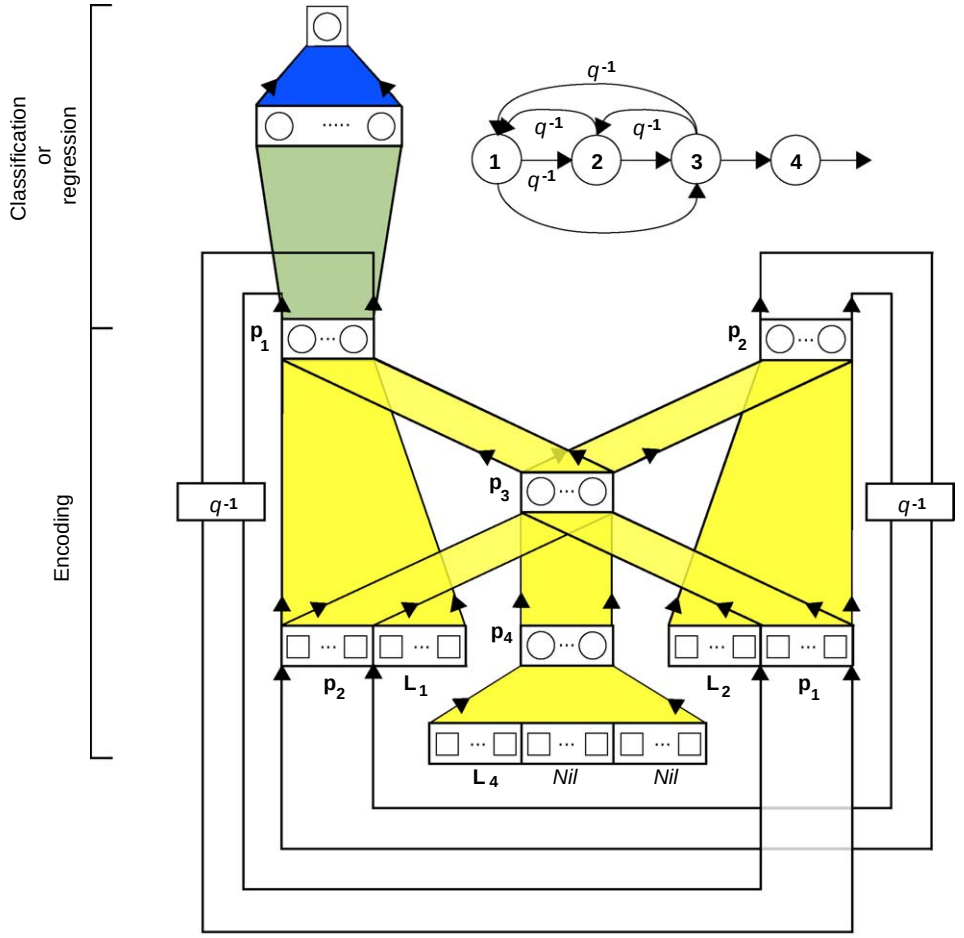


Fig. 16. A machine that can perform classification or regression with cyclic graphs. Four cycles are present in the graph, but two state vectors only are necessary.

part. The question was investigated in detail in [23–25]. The main results are summarized below.

Consider a mapping $F(x)$ of the set of k-ary trees, of height t, with real-valued labels, to $\mathbb{R}$. In the spirit of PAC-learning [56], we want to find a machine M, as described above, such that the probability of finding a tree x, whose image $F(x)$ differs from the machine's output $M(x)$ by more than a given ε, is smaller than a given δ:

$$\Pr[x \| M(x) - F(x)| > \varepsilon] < \delta. \tag{28}$$

The existence of such a machine was proved in [23–25]; it was also proved that the dimension of the codes, i.e. the size of the code vectors **p** as shown in the above figures, is equal to 2, if the labels are symbolic in nature. If the labels are real-valued from a compact set, then

the existence of a machine such that

$$|M(x) - F(x)| < \varepsilon \quad \text{for all } x \tag{29}$$

was proved. The dimension of the codes varies exponentially with the height t of the trees.

The universal approximation property has also been proved for cascade-correlation networks, which can be built incrementally [26].

Similarly to the universal approximation results for non-structured data, the above results are essentially existence theorems, which are very important since they provide firm ground on which applications can be built safely. However, whether a satisfactory machine can actually be found from real data depends strongly on implementation "details" such as the efficiency of the optimization algorithm. In addition, the assessment of the generalization ability of the machine thus found must be performed carefully, with statistically sound assessment methods [12].

5.5. Kernel methods for graphs

In the previous sections, we focused on non-parametric methods such as neural networks. Structured data can also be handled by Support Vector Machines or other kernel methods that requires the definition of "graph kernels" that measure the similarity between two graphs. The choice of the kernel function, which defines a "distance" between the examples, is an important issue. Due to the variety of graph structures, defining a kernel for general graphs has proven difficult; therefore, different kernels, for specific kinds of graphs, such as strings or trees, have been defined. Those kernels are all based on a vector of features counting the subelements (substrings, subtrees or paths) appearing in the graph.

5.5.1. Fisher kernels for tree-structured data

Fisher kernels were first used for the classification of biological sequences [32], but allow as well the comparison of trees with different topologies [52], by defining a vector representation whose dimension is independent of the size of the tree. We consider a Riemannian manifold M_θ defined by a class of parametric models $p_X(x|\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the vector of parameters of the family. Fisher kernels are based on Fisher scores, which are the gradients of the log likelihood of the observation with respect to the parameters $\boldsymbol{\theta}$:

$$\mathbf{u}_x = \nabla_{\boldsymbol{\theta}} \log\, p_X(x|\boldsymbol{\theta}). \tag{30}$$

Thus, the example x is described by the feature vector $\mathbf{u}_x$, which is the direction of maximum ascent of the log likelihood of x along the manifold. The kernel of this mapping, which measures the similarity of the graphs x and x', is then:

$$K(x, x') = \mathbf{u}_x^{\mathrm{T}} I^{-1} \mathbf{u}_{x'}. \tag{31}$$

The matrix $I = \mathrm{E}\{\mathbf{u}_x^{\mathrm{T}} \mathbf{u}_x\}$, where E is the expectation value, termed the Fisher information matrix, allows the normalization of the kernel. Moreover, this kernel is a valid definite positive kernel function.

In order to compute this kernel, the first step consists in estimating the parameters of the model $p_X(x|\boldsymbol{\theta})$ (a hidden tree Markov model) from a set of example trees. The Fisher

score of a tree x is computed from (30), allowing the computation of the Fisher kernel (31) between that tree and the example trees. The classification or regression is finally done with a SVM, or any kernel-based method.

5.5.2. String kernels

The first kernels developed for the handling of structured data were string kernels [37,38], which measure the similarity between two sequences.

We consider a sequence $x = (x_1, x_2, \ldots, x_{|x|})$, such that $x_i \in V = (v_1, \ldots, v_{|V|})$, where V is a finite set of symbols. The function $\mathbf{\Phi}^p$ is defined as

$$\mathbf{\Phi}^p(x) = \left[\varphi_1^p(x), \varphi_2^p(x), \ldots, \varphi_{|V^p|}^p(x)\right]^{\mathrm{T}}, \tag{32}$$

where $\varphi_u^p(x)$ is the number of occurrences of the subsequence u of length p in x. The *p-spectrum kernel* between two sequences x and y is then defined by the dot product:

$$K^p(x, y) = \langle \mathbf{\Phi}^p(x), \mathbf{\Phi}^p(y) \rangle = \sum_{u \in V^p} \varphi_u^p(x) \varphi_u^p(y). \tag{33}$$

In order to compare all the substrings that x and y have in common, the function $\mathbf{\Phi}$ that counts the substrings of all possible lengths p in x is defined:

$$\mathbf{\Phi}(x) = \left[\varphi_1(x), \varphi_2(x), \ldots, \varphi_{\bigcup_{1 \leqslant p \leqslant |x|} V^p}(x)\right]^{\mathrm{T}}. \tag{34}$$

Thus, x is mapped into a feature space $\mathbb{R}^d$, $d = \sum_{p=1}^{|X|} |V^p|$.

The string kernel of x and y is then the scalar product:

$$K(x, y) = \langle \mathbf{\Phi}(x) \cdot \mathbf{\Phi}(y) \rangle = \sum_{\substack{u \in V^p \\ 1 \leqslant p \leqslant \max(|X|, |Y|)}} \varphi_u^p(x) \varphi_u^p(y). \tag{35}$$

5.5.3. Tree kernels

In the same way as string kernels, tree kernels measure the similarity between two trees, by counting their identical constitutive sub-trees [8,27].

Let us consider two labeled trees T_1 and T_2, and the possible subtrees S_i, $1 \leqslant i \leqslant n$. Comparing T_1 and T_2 consists in counting the common subtrees they share, so that their similarity can be measured with their kernel defined by

$$K(T_1, T_2) = \sum_{i=1}^{n} H_i(T_1) H_i(T_2), \tag{36}$$

where $H_i(T_1)$ and $H_i(T_2)$ are the number of occurrences of the subtree S_i in T_1 and T_2, respectively.

A major drawback of this method is that some subtrees are given too much weight: the larger a common subtree, the more frequently it appears in the sum. Some alternatives to this definition were proposed to face this problem: they consist in downweighting the product

$H_i(T_1)H_i(T_2)$ in proportion to the size of the subtree H_i, so that:

$$K(T_1, T_2) = \sum_{i=1}^{n} \lambda^{\text{size}(H_i)} H_i(T_1)H_i(T_2) \quad \text{with } 0 < \lambda < 1. \tag{37}$$

5.5.4. Marginalized kernels between graphs

As opposed to trees, graphs can be cyclic and are not necessarily oriented, so that the number of paths in a graph can be infinite, and computing complete kernels, which requires counting all the possible paths in a graph, becomes unfeasible in terms of computational time. Several approaches have been proposed to address this problem, and find a trade-off between the efficiency of the computed kernel and the complexity of its computation [34,39].

Kernels for structured data are based on the detection of common paths between two graphs. Their definition involves "hidden variables" (subgraphs) H:

$$K(G, G') = \sum_{H} P(G|H)P(G'|H)P(H), \tag{38}$$

where G and G' are the structured data (graphs) and H is the hidden variable. Generally, $P(H|G)$ is known instead of $P(G|H)$, so that one computes

$$K(G, G') = \sum_{H}\sum_{H'} K_Z(Z, Z')P(H|G)P(H'|G'), \tag{39}$$

where $Z = [G, H]$, and $K_Z(Z, Z')$ is the joint kernel between Z and Z', which depends both on the hidden and visible variables. This kernel is known as the *marginalized kernel*.

As suggested by its expression, the marginalized kernel is the expectation of the joint kernel over all the possible values of H and H', that is, all the possible paths H and H', respectively, in G and G'. Let us consider the graph G, defined by its vertices V_i, $1 \leqslant i \leqslant |G|$, with labels v_i, and its edges E_{ij}, with labels e_{ij}. If we assume that the hidden variable H is obtained by a random walk of length l on G, it is a sequence of numbers, corresponding to vertices, ranging from 1 to $|G|$:

$$H = (h_1, h_2, \ldots, h_l) \tag{40}$$

and the probability of generating this sequence can be computed by making use of:

- the probability distribution of the starting point: $p_S(h_1)$
- the transition probability: $p_t(h_i|h_{i-1})$
- the probability distribution of the end point h_l: $p_e(h_l)$

such that $\sum_{j=1}^{|G|} p_t(h_j|h_i) + p_e(h_l) = 1$.

Therefore, the posterior probability of the sequence $H = (h_1, h_2, \ldots, h_l)$ is

$$P(H|G) = p_S(h_1) \cdot \prod_{i=2}^{l} p_t(h_i|h_{i-1})p_e(h_l) \tag{41}$$

p_S can be chosen as a uniform distribution, $p_t(h_i|h_{i-1})$ as a uniform distribution over the vertices adjacent to the current vertex h_{i-1} and p_e is a constant.

The joint kernel between $Z = [G, H]$ and $Z' = [G', H']$ must then be defined.

Since a random walk defines a sequence of vertices and edge labels:

$$(v_{h_1}, e_{h_1h_2}, \ldots, e_{h_{l-1}h_l}, v_{h_l})$$

the joint kernel between Z and Z' can be defined from two auxiliary kernels, between the sequences of vertices and edges, respectively:

$$K_Z(Z, Z') = \begin{cases} 0 & \text{if } l \neq l', \\ K(v_{h_1}, v'_{h'_1}) \prod_{i=2}^{l} K(e_{h_{i-1},h_i}, e'_{h'_{i-1},h'_i}) K(v_{h_i}, v'_{h'_i}) & \text{if } l = l', \end{cases} \tag{42}$$

where the kernels $K(v, v')$ and $K(e, e')$, which must be non-negative, can be defined in several ways. For example, one can assume that $K(v, v') = \delta(v, v')$, which implies that $K_Z(Z, Z') = 0$ as soon as H and H' differ by one vertex. $K(v, v')$ can also be the Gaussian kernel:

$$K(v, v') = \exp\left(\frac{-|v - v'|^2}{2\sigma^2}\right). \tag{43}$$

Such kernels have the advantage of guaranteeing the convergence of $K(G, G')$ even though the sum is a priori infinite.

However, the graph kernel thus defined has several drawbacks. First, its computation is time consuming. Furthermore, all the subpaths are given the same importance, whereas in specific studies, some paths may be more important than others.

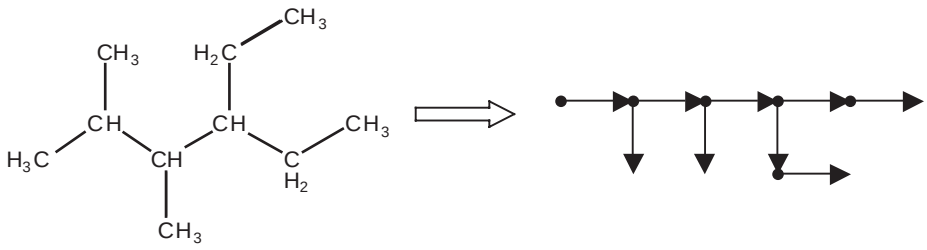
Once the graph kernels are defined, clustering, regression or classification can be performed in feature space, with appropriate tools such as SVMs, Perceptron, Nearest Neighbor or PCA.

6. Selected applications

In the present section, we describe applications of the methods described above for handling structured data, in the three most popular areas of applications: prediction of properties of molecules, image processing and natural language processing.

6.1. Prediction of properties or activities of molecules

The prediction of the properties or activities of molecules is an important issue in the chemical industry, for instance for computer-aided drug design. The major cost of a drug results from the fact that roughly 1 molecule out of 10,000 molecules that are synthesized and tested reaches the market. Therefore, it is important to be able to predict the activity of a molecule from its chemical structure, so as to avoid the cost of synthesizing molecules that do not have the desired property, or which have undesirable effects. Traditional machine learning techniques have been successfully used, during the past few years, in QSAR (quantitative structure-activity relations) and QSPR (quantitative structure–property relations), see for instance [16,60]. Those approaches first require the design and computation of the features (also termed descriptors) that are relevant for the prediction of the property of



2,3-dimethyl-4-ethylhexane

Tree representation of the molecule

Fig. 17. An alkane and its graph representation.

interest; they can be of different natures: geometrical, chemical, physical. Since the number of candidate features may be quite high, feature selection is mandatory, in order to select the features whose effect on the property of interest is larger than measurement noise or inaccuracy.

An alternative way of performing these tasks consists in considering the molecules as structured data, which allows the model designer to take advantage of the techniques described in the previous sections. Molecules can be considered as undirected graphs, by assigning each atom to a node and each bond to an edge; several approaches can be taken to build directed graphs from complex molecules featuring atoms of different natures, multiple bonds or cycles for example.

The prediction of the boiling point of alkanes is becoming a benchmark for comparisons between different methods. The data set features 150 molecules with up to 10 carbon atoms. These molecules are naturally represented by trees, by associating each carbon–hydrogen group to a node, each bond between these groups to an edge. These edges are directed with the choice of the first carbon–hydrogen group according to the IUPAC (International Union of Pure and Applied Chemistry) nomenclature system as the root of the tree. Since that representation follows specific rules, it is unique. Fig. 17 is an example of such a description.

Those methods are also able to handle more complex examples, and can perform tasks such as the prediction of the activity of benzodiazepines, but the representation of these molecules (see for example the molecule displayed in Fig. 18) as directed trees is less obvious, since they involve cycles and different kinds of bonds.

However, a major group, which includes the cycles, remains unchanged throughout this family of molecules, so that it can be used as a template, which is taken as the root of the tree whose branches depict the groups attached to the template.

The prediction of the property (i.e. boiling point) or activity of interest thus becomes a mapping of a set of trees T into $\mathbb{R}$; as described in Section 5, that can be performed by recursive networks or graph kernels (see for instance [2,42]).

As explained above, the task of representing molecules as directed trees can be obvious when dealing with basic examples, but can turn out to be difficult when cycles, multiple bonds or even aromatic cycles are involved. Graph machines [20] are able to handle such complex molecules, and allow their representation as directed acyclic graphs without any loss of information. This conversion requires to first consider the molecules as cyclic

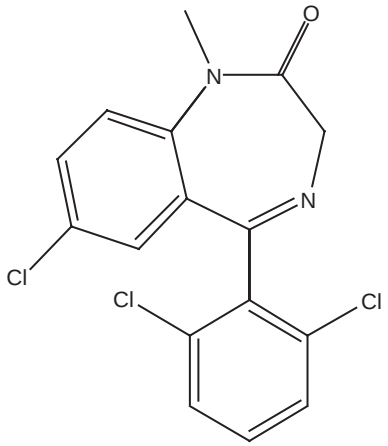


Fig. 18. A molecule involving cycles and double bonds.

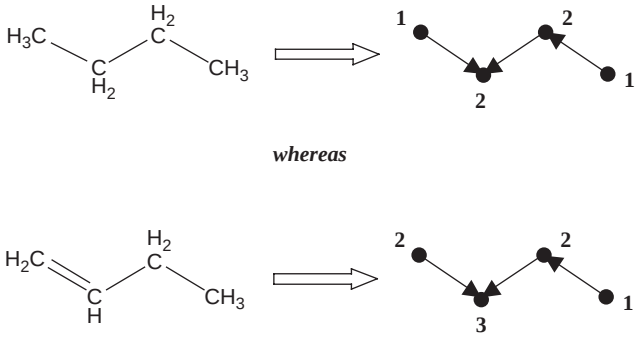


Fig. 19. Labeling each node with its degree preserves the information on the multiplicity of the bonds.

undirected graphs: each atom (other than hydrogen) is a node of the graph, labeled by the nature of the corresponding atom, its degree and other possible information (stereoisomery, electrical charge…); each bond, multiple or not, is an edge. The information on the multiplicity of the bonds is indeed preserved in the degrees of the nodes, as shown in Fig. 19.

The next step consists in ordering the nodes with specific rules (following for example the algorithm proposed in [33]), which allows the model designer to choose a root node for the graph, to select and cut as many edges as there are cycles in the graph, and to finally give the edges a direction, from the root of the tree to its terminal nodes. Even if the cut edges are no longer present in the directed acyclic graph (or DAG) formed in this way, the information on their presence is also saved due to the labels of the nodes. Fig. 20 shows how a molecule with aromatic cycles and heteroatoms can be represented as a DAG. Comparisons between graph machines, recursive networks, SVMs and conventional networks are reported in [20].

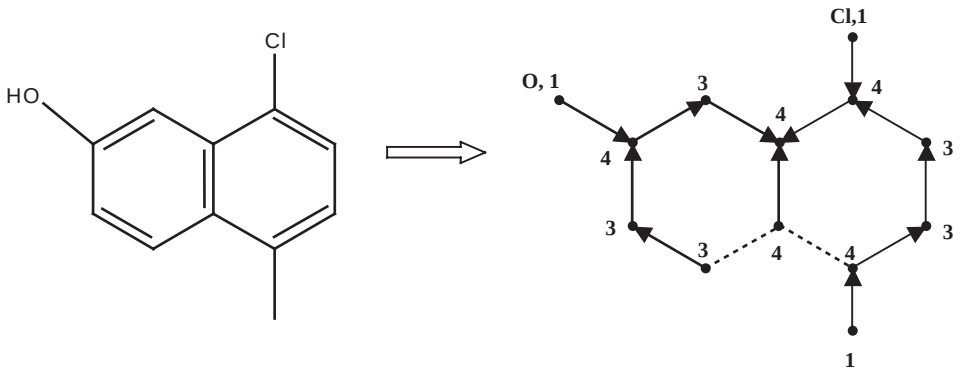


Fig. 20. A cyclic molecule represented as a DAG.

Structured data learning techniques have also been successfully used for investigating larger molecules, hence large-scale graphs, such as proteins [1].

6.2. Image analysis

Image analysis can be performed conveniently by first designing a high-level description of the image under the form of the graph, namely, a region adjacency graph. An algorithm segments the image into basic elements, which are assumed to have similar characteristics, for example to be uniformly colored or to have the same textures. Those elements are represented as the nodes of a graph, and are given labels encoding their relevant features, such as their shapes, sizes, orientations, and colors, in real-valued vectors. Each node is subsequently connected to the nodes corresponding to the adjacent elements (according to a pre-defined adjacency relationship), with edges that can be labeled, e.g. by the distance between the centroids of the regions. Thus, the image is represented as a graph preserving both its structural and sub-symbolic information. The undirected graph is then turned into a directed graph by choosing a root node and specific rules to order the nodes, and by directing the edges in that order [17].

The choice of the basic elements associated to the nodes depends on the properties of the processed images. For example, the classification of three kinds of images (airplanes, cars and weather maps) is performed in [19]. Those images are segmented into regions of homogeneous features with a region-growing algorithm. The regions are associated to nodes, whose labels encode the geometric and low-level (color, shape) features. Two nodes are subsequently connected if the corresponding regions share a border on the image. The same algorithm can be used to represent fingerprints [59]: a node corresponds to a region with homogeneous ridge directions, and is labeled with local features (area, orientation) and relations with the adjacent regions.

Other definitions of elements and adjacency relationships can be required to process different kinds of images. For example, in [11], the images to classify are logos, which are mainly made of contours, so that their structures representations can easily be extracted with the *contour tree algorithm*. The contours of the shapes in the image are extracted and

associated to the nodes of a graph, whose labels describe the properties of the contour (area, perimeter, or more elaborate features such as the average distance to the centroid along the contour). The nodes are then connected with edges following inclusion relationships between the contours. The graphs obtained can then be processed either by kernel methods, or by recursive networks.

6.3. Handling natural language problems

Graph methods are suitable for addressing natural language problems since texts can be represented as structured data. These problems include text interpretation, ambiguity resolution, or text categorization, and can be encountered when searching for relevant information on the web, for instance. Several approaches can be taken: string kernels are used in [38], whereas in [41], several graph trees are built for each sentence under consideration.

The traditional approach ("bag of words") in text classification consists in converting the text into a high-dimensional vector of features, where each value in this vector represents the presence or the absence of a feature (for example the frequency of each word in the text), thus losing all information on the ordering of the words. Alternatively, texts can be viewed as structured data, thereby allowing the use of graph methods.

The most obvious way of converting a text into a graph is to consider that a text is a sequence of strings [38], and to compare two different texts by comparing those sequences. To convert a text into a sequence of nodes, each string is considered as a node; in [38], a string is a single character, whose label identifies the character. To perform a classification or a clustering on a set of texts, kernel methods can be used. The degree of similarity between each pair of texts is measured with their string kernel, which is the inner product of their feature vectors as described in Section 5.5.2, and is the sum, over all common subsequences, of their frequency of occurrence weighted by their lengths

$$K(s_1, s_2) = \sum_{u \in \Sigma^n} \langle \phi_u(s_1) \cdot \phi_u(s_2) \rangle, \tag{44}$$

where Σ^n is the set of all substrings of length n, and $\phi_u(s_1) = \sum_{u=\text{subsequence}(S_1)} \lambda^{l(i)}$.

Several parameters can be adjusted, such as the maximal length n of the substrings, or the length of the possible gap in a non-contiguous substring. Once the string kernels are computed, clustering or classification can be carried out with traditional kernel methods.

Other methods can be used when more intricate problems arise, for example when trying to resolve ambiguities of understanding in a text. The "learning preferences" approaches can be taken to solve such problems, the task being to select the best representation of a text. That can be achieved by building a separate tree for each possible understanding, and assigning to each tree a probability of being the true representation of the text.

First-pass attachment problems are an example of syntactic ambiguity resolution tasks. They consist in assessing the relationships between groups of words, for example finding which noun a pronoun refers to in a sentence. A standard example [10] is that of finding out, in the sentence,

The servant of the actress who was on the balcony died.

whether "who" refers to "the servant" or to "the actress".

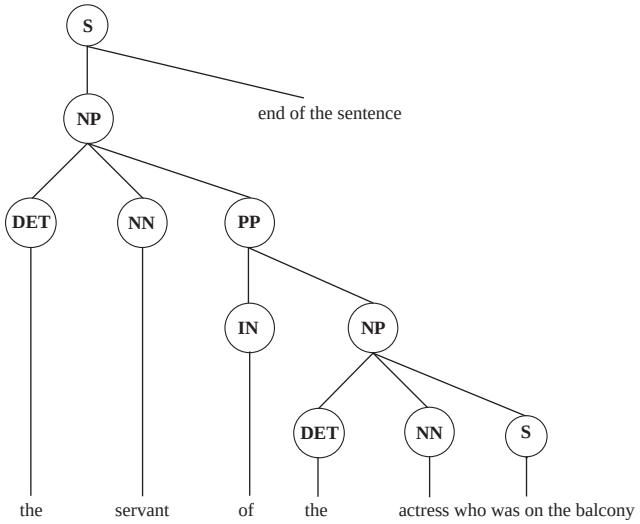


Fig. 21. A tree representation of a sentence.

Another problem is that of ranking possible parse or tag trees generated for the same sequence, i.e. finding the probabilities for each tree to correspond to the best representation of the sequence. Therefore, whereas in the text classification tasks a single tree is assigned to each example (text or sequence) and then classified, those problems require building, for each example, as many trees as possible interpretations, and assigning to each tree a score according to its probability of corresponding to the right representation.

A sentence is represented by a phrase structure tree, which is incrementally built as the sentence is read from left to right. Each new word i is connected to the tree T_{i-1} via an "anchor" word, which is the attachment word, to form the new tree T_i. The difference between the incremental trees T_{i-1} and T_i is called the *connection path*. The nodes of the tree are given tags, which can be *POS* (*part of speech*) *tags*, and correspond to the grammatical function of the single lexical item the node represents, or non-terminal tags, which characterize phrases. For instance, the tree shown in Fig. 21 can be built for the above sentence.

The labels DET, NN and IN are POS-tag labels, which represent a determinant, a noun and a preposition, respectively, whereas the labels NP and PP are non-terminal tags for Noun Phrases and Preposition Phrases. The attachment ambiguity problem appears when, for a new word, several connections (several anchors or connection paths) exist, which leads to the construction of several possible trees. For instance, the tree built for the ambiguous sentence corresponds to the interpretation for which the pronoun "who" refers to "the actress", but we obtain a different tree if we consider that "who" refers to "the servant". In order to find out which is the most likely to be the correct tree, each candidate tree is presented to a classifier that has been trained; in the training set, each example is a pair (F_i, j), where F_i designs a forest of possible trees for an example sentence, and j is the index of the correct tree. During the training step, the parameters of the regression tool (e.g. the weights of a neural network) are adjusted so that the highest probabilities are assigned to the correct trees, and

these parameters are then used to determine which interpretation of a new sentence is the correct one.

In [41], a first-pass attachment problem and a ranking task are performed. For each sequence of the first-pass attachment problem, 120 alternative incremental trees were built on average, while the ranking task is carried out on sentences for which an average of 30 alternative parse trees is proposed. RNNs and a kernel method were used and both lead to satisfying results.

7. Conclusion

The processing of structured data with numerical learning machines has been a long-standing problem, which attracted interest very early. In this short survey, we have shown that two basic ideas, often not explicitly stated, underlie the development of the methods:

- when structure is present in the data, a machine can be built, whose structure is the image of the structure of the data; recursive networks are essentially feedforward nets whose architecture is directly derived from the structure present in the data; that is reminiscent of semi-physical modeling, where the architecture of the network is derived from the architecture of prior knowledge on the process;
- the representation of structured data can be learnt instead of being handcrafted; learning the representation can be performed simultaneously with the learning of the task, whether classification or regression; that is reminiscent of convolutional networks.

Important existence theorems have been proven, giving a strong theoretical foundation to the approach. Moreover, we have presented typical successful applications, and more are to come.

There are still exciting problems that have been widely investigated for the learning of unstructured data, and should be considered carefully for structured data:

- *model selection and overfitting detection*: as usual in nonlinear black-box modeling, the complexity of the model should match the complexity of the data; therefore, the problem of the estimation of the generalization error is central in model design; although cross-validation extends trivially to structured data, alternative approaches should be investigated;
- *feature selection*: the nature of the problem to be solved may suggest several different sets of labels for the nodes; to the best of our knowledge, the choice of the best representation of the data, in terms of node labels, is an open problem;
- *example selection* ("*active learning*"): when faced with very large data sets, finding the most informative examples may be a crucial issue;
- *experimental planning*: when a small database is available, but additional (possibly costly) experiments can be performed, it is important to resort to principled methods for choosing the experiments that will be most important for improving the model; this issue is conceptually related to the previous one.

Acknowledgements

This work has been supported in part by CEMIP, and by a grant from the MENRT.

References

[1] P. Baldi, G. Pollastri, The principled design of large-scale recursive neural network architectures—DAG-RNNs and the protein structure prediction problem, J. Mach. Learning Res. 4 (2003) 576–602.
[2] A.M. Bianucci, A. Micheli, A. Sperduti, A. Starita, Application of cascade-correlation networks for structures to chemistry, Appl. Intelligence 12 (2000) 117–147.
[3] E. Bienenstock, R. Doursat, Elastic matching and pattern recognition in neural networks, in: L. Personnaz, G. Dreyfus (Eds.), Neural Networks: From Models to Applications, IDSET, Paris, 1989, pp. 472–482.
[4] E. Bienenstock, C. von der Malsburg, A neural network for invariant pattern recognition, Europhys. Lett. 4 (1987) 121–126.
[5] C. Bishop, Neural Networks for Pattern Recognition, Clarendon, Oxford, 1995.
[6] D.S. Blank, L.A. Meeden, J.B. Marshall, Exploring the symbolic/subsymbolic continuum: a case study of RAAM, in: J. Dinsmore (Ed.), The Symbolic and Connectionist Paradigms: Bridging the Gap, LEA Publishers, 1992.
[7] H.H. Chen, Y.C. Lee, T. Maxwell, C. Giles, High-order correlation model for associative memory, in: J.S. Denker (Ed.), Neural Networks for Computing, American Institute of Physics Conference Proceedings, Vol. 151, 1986, pp. 86–99.
[8] M. Collins, N. Duffy, Convolution kernels for natural language, in: S.B.T.G. Dietterich, Z. Ghahramani (Eds.), Advances in Neural Information Processing System, Vol. 14, MIT Press, Cambridge, MA, 2002, pp. 625–632.
[9] T. Cover, Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition, IEEE Trans. Electronic Comput. 14 (1965) 326–334.
[10] F. Cuetos, D.C. Mitchell, Cross-linguistic differences in parsing: restrictions on the use of the Late Closure strategy in Spanish, Cognition 30 (1988) 72–105.
[11] M. Diligenti, M. Gori, M. Maggini, E. Martinelli, Adaptive graphical pattern recognition for the classification of company logos, Pattern Recognition 34 (2001) 2049–2061.
[12] G. Dreyfus, Assessment methods, in: I. Guyon, S. Gunn, M. Nikravesh, L. Zadeh (Eds.), Feature Extraction, Foundations and Applications, Springer, Berlin, 2005.
[13] G. Dreyfus, Y. Idan, The canonical form of non-linear discrete-time models, Neural Comput. 10 (1998) 133–164.
[14] G. Dreyfus, J.M. Martinez, M. Samuelides, M. Gordon, F. Badran, S. Thiria, L. Hérault, Neural Networks, Methodology and Applications, Springer, Berlin, 2005.
[15] G. Dreyfus, A. Zippelius, Graph recognition by neural networks, in: L. Personnaz, G. Dreyfus (Eds.), Neural Networks from Models to Applications, IDSET, 1988, pp. 483–492.
[16] A. Duprat, T. Huynh, G. Dreyfus, Towards a principled methodology for neural network design and performance evaluation in QSAR; application to the prediction of logP, J. Chem. Inform. Comput. Sci. 38 (1998) 586–594.
[17] C. Goller, M. Gori, M. Maggini, Feature extraction from data structures with unsupervised recursive neural networks, Internat. Joint Conf. on Neural Networks, IEEE, 1999, pp. 1121–1126.
[18] C. Goller, A. Küchler, Learning task-dependent distributed structure-representations by backpropagation through structure, IEEE Internat. Conf. on Neural Networks, 1996, pp. 347–352.
[19] M. Gori, M. Maggini, L. Sarti, A recursive neural network model for processing directed acyclic graphs with labeled edges, Internat. Joint Conf. on Neural Networks, 2003, pp. 1351–1355.
[20] A. Goulon-Sigwalt-Abram, A. Duprat, G. Dreyfus, Learning numbers from graphs, Appl. Statist. Modeling Data Anal. (2005), available electronically from http://asmda2005.enst-bretagnefr/IMG/pdf/proceedings/552.pdf.
[21] I. Guyon, L. Personnaz, J.P. Nadal, G. Dreyfus, Storage and retrieval of complex sequences in neural networks, Phys. Rev. A 38 (1988) 6365–6372.
[22] I. Guyon, L. Personnaz, P. Siarry, G. Dreyfus, Engineering applications of spin glass concepts, in: J.L. van Hemmen, I. Morgenstern (Eds.), Heidelberg Colloquium on Glassy Dynamics, Lecture Notes in Physics, Vol. 275, Springer, Berlin, 1987.
[23] B. Hammer, On the approximation capability of recurrent neural networks, Neurocomputing 31 (2000) 107–123.

[24] B. Hammer, Learning with recurrent neural networks, Springer Lecture Notes in Control and Information Sciences, Vol. 254, Springer, Berlin, 2000.
[25] B. Hammer, Recurrent networks for structured data — A unifying approach and its properties, Cognitive Systems Res. 3 (2002) 145–165.
[26] B. Hammer, M. Alessio, A. Sperduti, Universal approximation capability of cascade correlation for structures, Neural Comput. 17 (2005) 1109–1159.
[27] D. Haussler, Convolution kernels on discrete structures, Technical Report UCSC-CLR- 99-10, University of Santa Cruz, 1999.
[28] J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities, Proc. Nat. Acad. Sci. 81 (1982) 2554–2558.
[29] J.J. Hopfield, D. Tank, "Neural" computation of decisions in optimization problems, Biol. Cybernet. 52 (1985) 141–152.
[30] K. Hornik, Some new results on neural network approximation, Neural Networks 6 (1993) 1069–1072.
[31] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (1989) 359–366.
[32] T.S. Jaakkola, D. Haussler, Exploiting generative models in discriminative classifiers, in: S.S.M. Kearns, D. Cohn (Eds.), Advances in Neural Information Processing Systems, Vol. 11, MIT Press, Cambridge, MA, 1998, pp. 487–493.
[33] C. Jochum, J. Gasteiger, Canonical numbering and constitutional symmetry, J. Chem. Inform. Comput. Sci. 17 (1977) 113–117.
[34] H. Kashima, K. Tsuda, A. Inokuchi, Marginalized kernels between labeled graphs, in: T. Faucett, N. Mishra (Eds.), 20th Internat. Conf. on Machine Learning, AAAI Press, Merlo Park, CA, 2003, pp. 321–328.
[35] R. Kree, A. Zippelius, Recognition of topological features of graphs and images in neural networks, J. Phys. A 21 (1988) 813–818.
[36] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, L.D. Jackel, Backpropagation applied to handwritten zip code recognition, Neural Comput. 1 (1989) 541–551.
[37] C. Leslie, E. Eskin, A. Cohen, J. Weston, W. Stafford Noble, Mismatch string kernels for discriminative protein classification, Bioinformatics 20 (2004) 467–476.
[38] H. Lodhi, C. Saunders, N. Cristianini, J. Shawe-Taylor, C. Watkins, Text classification using string kernels, J. Mach. Learning Res. 2 (2002) 419–444.
[39] P. Mahé, N. Ueda, T. Akutsu, J.-L. Perret, J.-P. Vert, Extensions of marginalized graph kernels, 21st Internat. Conf. on Machine learning, ACM Press, New York, 2004, 552–559.
[40] S. Marcos, O. Macchi, C. Vignat, G. Dreyfus, L. Personnaz, P. Roussel-Ragot, A unified framework for gradient algorithms used for filter adaptation and neural network training, Internat. J. Circuit Theory Appl. 20 (1992) 1159–1200.
[41] S. Menchetti, F. Costa, P. Frasconi, M. Pontil, Wide coverage natural language processing using kernel methods and neural networks for structured data, Pattern Recognition Lett. 26 (2005) 1896–1906.
[42] A. Micheli, F. Portera, A. Sperduti, A preliminary empirical comparison of recursive neural networks and tree kernel methods on regression tasks for tree structured domains, Neurocomputing 64 (2005) 73–92.
[43] O. Nerrand, D. Urbani, P. Roussel-Ragot, L. Personnaz, G. Dreyfus, Training recurrent neural networks: why and how? An illustration in process modeling, IEEE Trans. Neural Networks 5 (1994) 178–184.
[44] Y. Oussar, G. Dreyfus, How to be a gray box: dynamic semi-physical modeling, Neural Networks 14 (2001) 1161–1172.
[45] L. Personnaz, I. Guyon, G. Dreyfus, Information storage and retrieval in spin glass like neural networks, J. Physique Lettres 46 (1985) L 359–L 365.
[46] L. Personnaz, I. Guyon, G. Dreyfus, Collective computational properties of neural networks: new learning mechanisms, Phys. Rev. A 34 (1986) 4217–4228.
[47] L. Personnaz, I. Guyon, G. Dreyfus, High-order neural networks: information storage without errors, Europhys. Lett. 4 (1987) 863–867.
[48] J. Pollack, Recursive distributed representations, Artificial Intelligence 46 (1990) 77–106.
[49] W. Press, S.A. Teukolsky, Numerical Recipes, The Art of Scientific Computing, Cambridge University Press, Cambridge, 2002.
[50] B. Quenet, S. Sirapian, R. Dubois, G. Dreyfus, D. Horn, Modeling spatiotemporal olfactory data in two steps: from binary to Hodgkin–Huxley neurons, Biosystems 67 (2002) 203–211.

[51] F. Rosenblatt, The Perceptron: a probabilistic model for information storage and organization in the brain, Psychol. Rev. 65 (1956) 386–408.
[52] C. Saunders, J. Shawe-Taylor, A. Vinokourov, String kernels, Fisher kernels and finite state automata, in: S. Becker, S. Thrun, K. Obermayer (Eds.), Advances in Neural Information Processing Systems, Vol. 15, MIT Press, Cambridge, MA, 2003, pp. 633–640.
[53] H.T. Siegelmann, E.D. Sontag, On the computational power of neural networks, J. Comput. System Sci. 50 (1995) 132–150.
[54] E.D. Sontag, Neural networks for control, in: Essays on Control: Perspectives in the Theory and its Applications, Birkhäuser, Basel, 1993, pp. 339–380.
[55] A. Sperduti, Encoding labeled graphs by labeling RAAM, Connection Sci. 6 (1994) 429–459.
[56] L.G. Valiant, A theory of the learnable, Commun. ACM (1984) 1134–1142.
[57] C. von der Malsburg, E. Bienenstock, A neural network for the retrieval of superimposed patterns, Europhys. Lett. 3 (1987) 1243–1249.
[58] C. Waibel, T. Hanazawa, G. Hinton, K. Shikano, K. Lang, Phoneme recognition using time-delay neural networks, IEEE Trans. Acoust., Speech Signal Process. 37 (1989) 328–339.
[59] Y. Yao, G.L. Marcialis, M. Pontil, P. Frasconi, F. Roli, Combining flat and structured representations for fingerprint classification with recursive neural networks and support vector machines, Pattern Recognition 36 (2003) 397–406.
[60] J. Yao, A. Panaye, J.P. Doucet, R.S. Zhang, H.F. Chen, M.C. Liu, Z.D. Hu, B.T. Fan, Comparative study of QSAR/QSPR correlations using support vector machines, radial basis function neural networks, and multiple linear regression, J. Chem. Inform. Comput. Sci. 44 (2004) 1257–1266.