ELSEVIER

Artificial Intelligence 125 (2001) 155–207

Artificial Intelligence

www.elsevier.com/locate/artint

Symbolic knowledge extraction from trained neural networks: A sound approach

A.S. d'Avila Garcez [a,*], K. Broda [a], D.M. Gabbay [b]

[a] *Department of Computing Imperial College, London SW7 2BZ, UK*

[b] *Department of Computer Science, King's College London WC2R 2LS, UK*

Received 15 August 1999; received in revised form 29 June 2000

Abstract

Although neural networks have shown very good performance in many application domains, one of their main drawbacks lies in the incapacity to provide an explanation for the underlying reasoning mechanisms.

The "explanation capability" of neural networks can be achieved by the extraction of symbolic knowledge. In this paper, we present a new method of extraction that captures nonmonotonic rules encoded in the network, and prove that such a method is sound.

We start by discussing some of the main problems of knowledge extraction methods. We then discuss how these problems may be ameliorated. To this end, a partial ordering on the set of input vectors of a network is defined, as well as a number of pruning and simplification rules. The pruning rules are then used to reduce the search space of the extraction algorithm during a pedagogical extraction, whereas the simplification rules are used to reduce the size of the extracted set of rules. We show that, in the case of regular networks, the extraction algorithm is sound and complete.

We proceed to extend the extraction algorithm to the class of non-regular networks, the general case. We show that non-regular networks always contain regularities in their subnetworks. As a result, the underlying extraction method for regular networks can be applied, but now in a decompositional fashion. In order to combine the sets of rules extracted from each subnetwork into the final set of rules, we use a method whereby we are able to keep the soundness of the extraction algorithm.

Finally, we present the results of an empirical analysis of the extraction system, using traditional examples and real-world application problems. The results have shown that a very high fidelity between the extracted set of rules and the network can be achieved.

Keywords: Neural-symbolic integration; Rule extraction; Nonmonotonic reasoning; Artificial neural networks

* Corresponding author.

E-mail addresses: aag@doc.ic.ac.uk (A.S. d'Avila Garcez), kb@doc.ic.ac.uk (K. Broda), dg@dcs.kcl.ac.uk (D.M. Gabbay).

PII: S0004-3702(00)00077-1

1. Introduction

Human cognition successfully integrates the connectionist and symbolic paradigms of Artificial Intelligence (AI). Yet, the modelling of cognition develops these separately in neural computation and symbolic logic/AI areas. There is now a movement towards a fruitful midway in between these extremes, in which the study of logic is combined with recent insights from connectionism. It is essential that these be integrated [22].

The aim of neural-symbolic integration is to explore the advantages that each paradigm presents. Within the features of artificial neural networks are massive parallelism, inductive learning and generalization capabilities [7,13]. On the other hand, symbolic systems can explain their inference process, e.g., through automatic theorem proving, and use powerful declarative languages for knowledge representation [17,19].

The Connectionist Inductive Learning and Logic Programming (CIL^2P) system [5] is a proposal towards tightly coupled neural-symbolic integration, which is best instantiated in [12] (see [14] for a classification of systems of neural-symbolic integration). CIL^2P is a massively parallel computational model based on a feedforward artificial neural network that integrates inductive learning from examples and background knowledge [18] with deductive learning from Logic Programming [19]. Starting with the background knowledge represented by a (propositional) general or extended logic program, a translation algorithm (see Fig. 1(1)) is applied generating a neural network that can be trained with examples (2). Moreover, the neural network computes the stable model (answer set) of the general (extended) program inserted in it or learned by examples, as a parallel system for Logic Programming (3). The final stage of the system (4) consists of the symbolic knowledge extraction from the trained neural network, which provides the explanation for the network's answers. The knowledge extracted then could feed the system again (5), closing the learning cycle. [1]

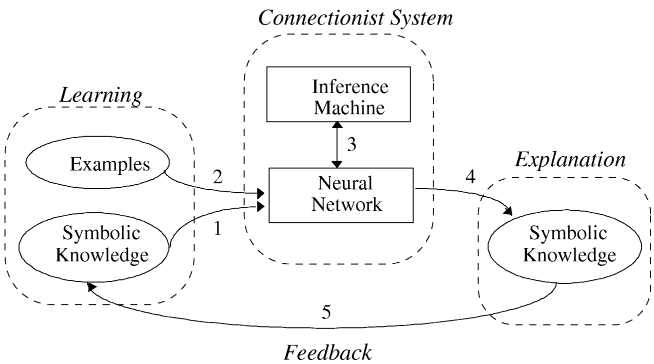


Fig. 1. Neural-symbolic integration.

[1] For example, in a fault diagnosis system, a neural network can detect a fault quickly, triggering safety procedures, while the knowledge extracted from it can justify the fault later on. If mistaken, this information can be used to fine tune the learning system.

In this paper, we concentrate on the problem of extraction of symbolic knowledge from trained neural networks, that is, the problem of finding "logical representations" for such networks. The extraction allows for the explanation of the decision making process, thus contributing to solve the "knowledge acquisition bottleneck problem". The domain theory extracted, obtained from inductive learning with examples, can be added to an existing knowledge-base or used in the solution of analogous domains problems.

Briefly, the problem of extraction lies on the complexity of the extraction algorithm. Holldobler and Kalinke [15] have shown that each logic program is equivalent to a single hidden layer neural network. In one direction of that equivalence relation, a translation algorithm (see Fig. 1(1)) derives a neat neural network structure when a logic program is given. The problem arises in the converse direction, i.e., given a trained neural network, how could we find out the equivalent logic program? Unfortunately, it is very unlikely that a neat network will result from the learning process. Furthermore, a typical real-world application network may contain hundreds of input neurons and thousands of connections.

The knowledge acquired by a neural network during its training phase is encoded as:

(i) the network's architecture itself;
(ii) the activation function associated to it; and
(iii) the value of its weights.

As pointed out in [2], the task of extracting explanations from trained neural networks is the one of interpreting in a comprehensible form the collective effect of (i), (ii), and (iii). Also in [2], a classification scheme for extraction algorithms is given, based on:

(a) the expressive power of the extracted rules;
(b) the "translucency" of the network;
(c) the quality of the extracted rules; and
(d) the algorithmic complexity.

The first classification item refers to the symbolic knowledge presented to the user from the extraction process. In general, this knowledge is represented by rules of the form "if then else". The second classification item contains two basic categories: decompositional and pedagogical. In the *decompositional*, the extraction occurs at the level of individual, hidden and output, units within the trained neural network. In the *pedagogical*, the neural network is viewed as a "black box", and the extraction is done by mapping inputs directly into outputs. The next classification item intends to measure how well the task of extracting the rules has been performed, considering the accuracy, consistency and comprehensibility of the set of rules. The last item refers to the requirement for the algorithm to be as effective as possible. In this sense, a crucial issue in developing an extraction algorithm is how to constrain its search space.

In [33], Thrun defines the following desirable properties of an extraction algorithm:

(i) no architectural requirements: a general extraction mechanism should be able to operate with all types of neural networks;
(ii) no training requirements: the algorithm should not make assumptions about the way the network has been built and how its weights and biases have been learned;
(iii) correctness: the extracted rules should describe the underlying network as correctly as possible;
(iv) high expressive power: more powerful languages and more compact rule sets are highly desirable.

Intuitively, the extraction task is to find the relations between input and output concepts in a trained network, in the sense that certain inputs *cause* a particular output. We argue that neural networks are nonmonotonic systems, i.e., they jump to conclusions that might be withdrawn when new information is available [21]. Thus, the set of rules extracted may contain default negation ($\sim$). Each neuron can represent a concept or its "classical" negation ($\neg$). Consequently, we expect to extract a set of rules of the form: $L_1, \ldots, L_n, \sim L_{n+1}, \ldots, \sim L_m \rightarrow L_{m+1}$, where each L_i is a literal (a propositional variable or its "classical" negation), L_j $(1 \leqslant j \leqslant m)$ represents a neuron in the network's input layer, L_{m+1} represents a neuron in the network's output layer, $\sim$ stands for default negation, and $\rightarrow$ means causal implication [2] (see [5] for neural network's nonmonotonic semantics).

In this paper, we present a new approach for knowledge extraction from trained networks that complies with the above perspective. We start by discussing some of the main problems found in the literature. We then discuss how these problems may be ameliorated. To this end, we identify a partial ordering on the set of input vectors of a network, and define a number of pruning rules and simplification rules that interact with such an ordering. These rules are used to reduce the search space of the extraction algorithm, as well as the number of rules extracted. We show that, in the case of regular networks, the extraction algorithm is sound and complete. [3] We then extend the extraction algorithm to the general case. By showing that every non regular network contains regularities in its subnetworks, we can still apply the underlying extraction algorithm to the general case network, but now in a decompositional fashion. The only problem we have to tackle, however, is how to combine the sets of rules obtained from each subnetwork into the set of rules of the network. We use a method for assembling the set of rules whereby we are able to preserve soundness of the extraction algorithm, although we have to forego completeness.

In Section 2, we discuss the main problems of the task of extracting knowledge from trained networks. In Section 3, we recall some useful preliminary concepts and define the extraction problem precisely. In Section 4, we present our solution to the extraction problem, culminating with the outline of the extraction algorithm for the class of regular networks, and the proofs of soundness and completeness of the method. In Section 5, we extend the extraction algorithm to the class of non regular networks—the general case—and show that the method of extraction is sound in this case. In Section 6, we present the experimental results of applying the extraction system to the Monk's Problems [32], DNA sequence analysis and Power Systems fault diagnosis. Finally, in Section 7, we conclude and discuss directions for future work.

2. Related work

Among the existing extraction methods, the one presented in [15], the "Ruleneg" [26], the "VIAnalysis" algorithm [33], and the "Rule-Extraction-as-Learning" method [8] use

[2] Notice that this is the language of Extended Logic Programming [11].

[3] Following [10], we say that an extraction algorithm is sound and complete if the set of rules is provably equivalent to the network. If, however, the set of rules is correct, but represents only a subset of the set of answers of the network, then the extraction is sound but incomplete.

"pedagogical" approaches, while the "Subset" [10], the "MofN" [35], the "Rulex" [3] and Setiono's proposal [29,30] are "decompositional" methods (see [2] for a comprehensive survey).

In the *CIL*^{2}*P* system, after learning takes place, the network N encodes a knowledge P' that contains the background knowledge P complemented or even revised by the knowledge learned with training examples. We want to derive P' from N. At the moment, only pedagogical approaches can guarantee that the knowledge extracted is equivalent to the network, i.e., that the extraction process is sound and complete. In [15], for instance, all possible combinations of the input vector $\boldsymbol{i}$ of N are taken into account in the process of rule generation. In this way, the method must consider 2^n different input vectors, where n is the number of neurons in the input layer of N. Some pedagogical approaches tackle this problem by extracting rules for the learning set only, excluding the network's generalization.

Obviously, pedagogical approaches are not effective when the size of the neural network increases, as in real-world applications. In order to overcome this limitation, decompositional methods, in general, apply heuristically guided searches to the process of extraction. The "Subset" method [10], for instance, attempts to search for subsets of weights of each neuron in the hidden and output layers of N, such that the neurons' input potential exceeds its threshold. Each subset that satisfies the above condition is written as a rule. One of the most interesting decompositional methods is the "MofN" technique [35]. Based on the Subset method, it uses weights' clustering and pruning in order to facilitate the extraction of rules. It also generates a smaller number of rules, by taking advantage of the *M of N* representation, in which $m(A_1, \ldots, A_n) \rightarrow A$ indicates that if m *of* $(A_1, \ldots, A_n)$ are *true* then A is *true*, where $m \leqslant n$. The work by Setiono [29,30] is another proposal of decompositional extraction. Setiono proposes a penalty function for pruning a feedforward neural network, and then generates rules from the pruned network by considering a small number of activation values at the hidden units.

Decompositional methods, such as [35] and [30], in general use weights pruning mechanisms prior to extraction. However, there is no guarantee that a pruned network will be equivalent to the original one. That is the reason why these methods usually require retraining the network. During retraining, some restrictions must be imposed on the learning process—for instance, allowing only the thresholds, but not the weights, to change—in order to the network to keep its "well-behaved" pruned structure. At this point, there is no guarantee that retraining will be successful under such restrictions. Other extraction methods use penalty functions during training to try and keep the initial "well-behaved" structure of the network and, thus, facilitate extraction. Such methods are bound to restrict the network's learning capability, as they would not be applicable to a network trained with an "*off the shelf*" learning algorithm. Even if we avoid the use of penalty functions and weights' clustering and pruning, the simple task of decomposing the network into smaller subnetworks, from which rules are extracted and then assembled, has to be carried out carefully. That is because, in general, the collective effect of the network is different from the effect of the superposition of its parts [2]. As a result, most decompositional methods are unsound. The following example illustrates this fact.

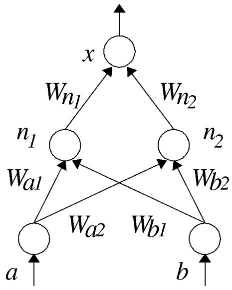


Fig. 2. A fully-connected network with two input neurons (a, b), two hidden neurons (n_1, n_2) and a single output neuron (x).

Example 1 (*Unsoundness and incompleteness of decompositional extraction algorithms*). Consider the network N of Fig. 2. Let us assume that the weights are such that $a = 1$ and $b = 1$ neither activate n_1 nor n_2, but that the composition of the activation values of n_1 and n_2 activates x. As a result, we would expect to extract $ab \to x$ from N. For example, suppose that $a = 1$ and $b = 1$ gives $n_1 = 0.3$ and $n_2 = 0.4$, and that these activation values result in $x = 0.99$. [4] A decompositional method would most probably derive a unique rule from such a network, namely, $n_1, n_2 \to x$, not being able to establish the correct relation between a and b, and x, that is, $ab \to x$.

Now, assume that inputs $a = 1$ and $b = 1$ activate n_1 and n_2, but n_1 and n_2 together do not activate x (say, $x < 0.5$). For example, assume $W_{a1} = 0.2$, $W_{b1} = 0.2$, $W_{a2} = 0.4$, $W_{b2} = 0.45$, $W_{n_1} = 9$ and $W_{n_2} = -8.1$, and take $f(x) = 1/(1 + \mathrm{e}^{-x})$ as the activation function of the neurons of N. Also, assume that the thresholds of neurons n_1, n_2 and x are all *zero*. In this case, $a = 1$ and $b = 1$ makes $n_1 \simeq 0.6$ and $n_2 \simeq 0.7$, which, in turn, outputs $x \simeq 0.4$. As a result, now we do not want to extract the rule $ab \to x$ from N. However, if n_1 and n_2 are approximated as threshold units then $n_1 = 1$ and $n_2 = 1$ produces $x \simeq 0.7$. In other words, although $a = 1$ and $b = 1$ does not activate x, approximating the sigmoidal activation function of n_1 and n_2 by a step function results in x being activated. Hence, decompositional methods that do so, such as [35], would conclude that $ab \to x$ when, in fact, $ab \nrightarrow x$.

The first of the above cases is an example of incompleteness. The second one shows how decompositional methods may turn out to be unsound. Even Fu's extraction [10], which is sound with respect to each hidden and output neuron, may become unsound with respect to the whole network due to the assumption that the activation function of the hidden neurons can be approximated by a step function.

Clearly, the classification of rule extraction methods as *pedagogical* or *decompositional* reflects a trade-off between the *complexity* of the extraction method and the *quality* of the knowledge extracted. In general, highly accurate, pedagogical methods of extraction

[4] For example, if $f(x) = 1/(1 + \mathrm{e}^{-x})$ is the activation function of the neurons in N, and the thresholds of n_1, n_2 and x are all *zero*, then $W_{a1} = -0.5$, $W_{b1} = -0.35$, $W_{a2} = -0.2$, $W_{b2} = -0.2$, $W_{n_1} = 3$ and $W_{n_2} = 9.25$ is a set of weights that makes N behave as intended for inputs $a = 1$ and $b = 1$.

present exponential complexity, while, more efficient, decompositional methods of extraction are unsound, and thus, have unpredictable accuracy, which can only be evaluated empirically in a particular application domain. In our view, an alternative is to prune the set of input vectors, rather than the set of weights, of the network from which we want to extract rules. Our goal is to reduce complexity in the average case by applying the extraction algorithm on a smaller search space, yet maintaining the highest possible quality, in particular to maintain soundness.

Differently from the above approaches, we also want to capture nonmonotonic rules encoded in the network. In order to do so, we add negation by default ($\sim$) to the language. We argue that one cannot derive a sensible set of rules from a network without having $\sim$ in the language, as the following example illustrates.

Example 2 (*Nonmonotonicity of neural networks*). Consider the neural network N of Fig. 3. Let $W_{n_1a} = 5$, $W_{n_1b} = -5$ and $W_{xn_1} = 1$. Assume that the activation function of a and b is the identity function $f(x) = x$, the activation function of n_1 and x is the standard sigmoidal function $h(x) = 1/(1 + \mathrm{e}^{-x})$, and let $\theta_{n_1} = \theta_x = 0.5$, where θ_{n_1} and θ_x are the thresholds of neurons n_1 and x, respectively. As a result, inputs $a = 1$ and $b = 0$ activate x $(x > 0.5)$. If one concludes, from that, that $a \to x$, one should be able to conclude as well that $ab \to x$, since the latter rule is subsumed by the former. However, inputs $a = 1$ and $b = 1$ do not activate x ($x < 0.5$ in N). In this case, one would conclude that $ab \nrightarrow x$, a contradiction!

Therefore, the correct rule to be extracted in the first place, when $a = 1$ and $b = 0$ activate x, is $a{\sim}b \to x$. The meaning of such a rule should be: x fires in the presence of a, provided that b is not present. In fact, if b turns out to be *true* then the conclusion of x is overruled, because $ab \nrightarrow x$. Such a nonmonotonic behavior should be captured by the extraction of rules with default negation ($\sim$), as opposed to classical negation ($\neg$), which is logically stronger than $\sim$ in the sense that a literal should be *proved*, instead of *assumed* by default. Classical negation should be explicitly represented in the network by a neuron labelled $\neg x$ (see [4]), as we will exemplify later in Section 6 with the experiments on knowledge extraction from a network that detects faults in a power plant. Thus, for the network N of Fig. 3, we should have $a{\sim}b \to x$ because x will be derived by N if a is

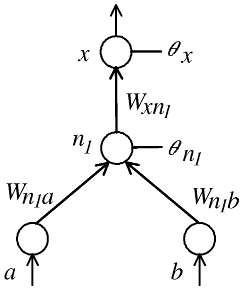


Fig. 3. A fully-connected network with two input neurons (a, b), a single hidden neuron (n_1) and a single output neuron (x).

added into N and b is assumed *false* by default. If b is also added into N then x will not be derived by N any longer. [5]

An immediate result of the above observation is that, in order to conclude that a network N'' with two input neurons, say a and b, encodes the rule $a \to x$, firstly we need to make sure that the following rules: $ab \to x$ and $a{\sim}b \to x$, are both encoded in N''. In other words, $a \to x$ should be seen as a *simplification* of the rules $ab \to x$ and $a{\sim}b \to x$ of N'', which indicate that b is a '*don't care*'. In this scenario, the use of *zeros* as input values could be misleading, as for example, when $a = 1$ and $b = 0$ led us to conclude that $a \to x$ could be a rule of N. For this reason, we find the use of $\{-1, 1\}$ inputs more appropriate (see also [5] for more on this subject).

Summarizing, the novelties on this paper are: we present an eclectic approach whereby we can reduce the complexity of the extraction algorithm in some interesting cases, yet executing a sound extraction, which we believe should be the minimum requirement of any method of rule extraction, and we capture nonmonotonicity in the set of rules extracted from the network, by adding default negation to the language.

3. Preliminaries

3.1. General

We need to assert some basic assumptions that will be used throughout this paper. $\mathbb{N}$ and $\mathbb{R}$ will denote the sets of natural and real numbers, respectively.

Definition 3. A *partial order* is a reflexive, transitive and antisymmetric relation on a set.

Definition 4. A partial order $\preceq$ on a set X is *total* iff for every $x, y \in X$, either $x \preceq y$ or $y \preceq x$. Sometimes, $\preceq$ is also called a *linear order*, or simply a *chain*.

As usual, $x \prec y$ abbreviates $x \preceq y$ and $x \neq y$.

Definition 5. In a partially ordered set $[X, \preceq]$, x is the *immediate predecessor* of y if $x \prec y$ and there is no element z in X such that $x \prec z \prec y$. The inverse relation is called the *immediate successor*.

Definition 6. Let X be a set and $\preceq$ an ordering on X. Let $x \in X$.

- x is *minimal* if there is no element $y \in X$ such that $y \prec x$.
- x is a *minimum* if for all elements $y \in X$, $x \preceq y$. If $\preceq$ is also antisymmetric and such an x exists, then x is unique and will be denoted by $\inf(X)$.
- x is *maximal* if there is no element $y \in X$ such that $x \prec y$.

[5] When a network N' encodes $a\neg b \to x$ then x is derived by N' only when a and $\neg b$ are added into it. In this case, if b is added as well then there is a contradiction in N', with b and $\neg b$, and, in Classical Logic, x would still be derived. From this, one sees that $\sim$ is required in the extraction of rules.

- x is a *maximum* if for all elements $y \in X$, $y \preceq x$. If $\preceq$ is also antisymmetric and such an x exists, then x is unique and will be denoted by $\sup(X)$.

A maximum (minimum) element is also maximal (minimal) but is, in addition, comparable to every other element. This property and antisymmetry leads directly to the demonstration of the uniqueness of $\inf(X)$ and $\sup(X)$.

3.2. Neural networks

Hornik, Stinchcombe and White [16] have proved that standard feedforward neural networks with a single hidden layer are capable of approximating any (Borel measurable) function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. Thus, we concentrate on single hidden layer networks, without loss of generality.

Given a single hidden layer feedforward network, the following systems of equations describe it.

$$\begin{aligned}
n_1 &= h\big(W^1_{11} i_1 + W^1_{12} i_2 + \cdots + W^1_{1p} i_p - \theta_{n_1}\big) \\
n_2 &= h\big(W^1_{21} i_1 + W^1_{22} i_2 + \cdots + W^1_{2p} i_p - \theta_{n_2}\big) \\
&\vdots \\
n_r &= h\big(W^1_{r1} i_1 + W^1_{r2} i_2 + \cdots + W^1_{rp} i_p - \theta_{n_r}\big),
\end{aligned} \tag{1}$$

$$\begin{aligned}
o_1 &= h\big(W^2_{11} n_1 + W^2_{12} n_2 + \cdots + W^2_{1r} n_r - \theta_{o_1}\big) \\
o_2 &= h\big(W^2_{21} n_1 + W^2_{22} n_2 + \cdots + W^2_{2r} n_r - \theta_{o_2}\big) \\
&\vdots \\
o_q &= h\big(W^2_{q1} n_1 + W^2_{q2} n_2 + \cdots + W^2_{qr} n_r - \theta_{o_q}\big),
\end{aligned} \tag{2}$$

where $\boldsymbol{i} = (i_1, i_2, \ldots, i_p)$ is the network's input vector ($i_{j(1 \leqslant j \leqslant p)} \in [-1, 1]$), $\boldsymbol{o} = (o_1, o_2, \ldots, o_q)$ is the network's output vector ($o_{j(1 \leqslant j \leqslant q)} \in [-1, 1]$), $\boldsymbol{n} = (n_1, n_2, \ldots, n_r)$ is the hidden layer vector ($n_{j(1 \leqslant j \leqslant r)} \in [-1, 1]$), $\theta_{n_j(1 \leqslant j \leqslant r)}$ is the jth hidden neuron threshold ($\theta_{n_j} \in \mathbb{R}$), $\theta_{o_j(1 \leqslant j \leqslant q)}$ is the jth output neuron threshold ($\theta_{o_j} \in \mathbb{R}$), $-\theta_{n_j}$ (respectively $-\theta_{o_j}$) is called the bias of the jth hidden neuron (respectively output neuron), $W^1_{ij(1 \leqslant i \leqslant r, 1 \leqslant j \leqslant p)}$ is the weight of the connection from the jth neuron in the input layer to the ith neuron in the hidden layer ($W^1_{ij} \in \mathbb{R}$), $W^2_{ij(1 \leqslant i \leqslant q, 1 \leqslant j \leqslant r)}$ is the weight of the connection from the jth neuron in the hidden layer to the ith neuron in the output layer ($W^2_{ij} \in \mathbb{R}$), and finally $h(x) = 2/(1 + \mathrm{e}^{-\beta x}) - 1$ is the standard bipolar (semi-linear) activation function. [6]

We define the extraction problem as follows:

[6] Whenever it is not necessary to differentiate between hidden and output layer, we refer to the weights in the network as W_{ij} only. Similarly, we refer to the network's thresholds in general as θ_i only.

Given a particular set of weights and thresholds θ_i, resulting from a training process on a neural network, find for each input vector $\boldsymbol{i}$, all the outputs o_j in the corresponding output vector $\boldsymbol{o}$ such that the activation of o_j is greater than A_{min}, where $A_{min} \in (0,1)$ is a predefined value (in this case, we say that output neuron o_j is "active" for input vector $\boldsymbol{i}$).

We assume that for each input i_j in the input vector $\boldsymbol{i}$, either $i_j = 1$ or $i_j = -1$. That is done because we associate each input (and output) neuron with a concept, say a, and $i_j = 1$ means that a is *true* while $i_j = -1$ means that a is *false*. For example, consider a network with input neurons a and b. If $\boldsymbol{i} = (1, -1)$ activates output neuron c then we derive the rule $a \sim b \to c$. As a result, if the input vector $\boldsymbol{i}$ has length p, there are 2^p possible input vectors to be checked.

4. The extraction algorithm for regular networks

Having identified the problems of knowledge extraction from trained networks, let us now start working towards the outline of their solutions. Given the above extraction problem definition, firstly we realize that each output neuron o_j has a constraint $\boldsymbol{C}_{o_j}$ associated. We want to find the activation value of o_j, $Act(o_j) = h(\sum_{i=1}^{r}(W_{ji}^2 n_i) - \theta_{o_j})$, such that $Act(o_j) > A_{min}$. Considering the monotonically crescent characteristic of the activation function $h(x)$ and given that $0 < A_{min} < 1$ and $\beta > 0$, we can rewrite $h(x) > A_{min}$ as $x > h^{-1}(A_{min})$. Hence, each output o_j is determined by the system of equations (1) above and Eq. (3) below, which is given in terms of the hidden neurons' activation values. [7]

$$o_j \text{ is active for } \boldsymbol{i} \text{ iff } W_{j1}^2 n_1 + W_{j2}^2 n_2 + \cdots + W_{jr}^2 n_r > h^{-1}(A_{min}) + \theta_{o_j}. \tag{3}$$

4.1. Positive networks

We start by considering a very simple network where all weights are positive real numbers. As a result, given two input vectors $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$, if for all i, $1 \leqslant i \leqslant r$, $n_i(\boldsymbol{i}_m) > n_i(\boldsymbol{i}_n)$ then for all j, $1 \leqslant j \leqslant q$, $o_j(\boldsymbol{i}_m) > o_j(\boldsymbol{i}_n)$, where $n_i(\boldsymbol{i})$ and $o_j(\boldsymbol{i})$ denote, respectively, the activation values of hidden neuron n_i and output neuron o_j, given input vector $\boldsymbol{i}$. Moreover, if $\boldsymbol{i}_m = (1, 1, \ldots, 1)$, the activation value of each neuron n_i is maximum and, therefore, the activation value of each neuron o_j is maximum as well. Similarly, if $\boldsymbol{i}_n = (-1, -1, \ldots, -1)$ then the activation of each n_i is minimum and, thus, so is the activation of each o_j. That results also from the monotonically crescent characteristic of the activation function $h(x)$, as we will see in detail later. Let us firstly present a simple example to help clarify the above ideas.

Example 7. Consider the network N of Fig. 4(1) and its associated constraint of Fig. 4(2). We know that $n_1 = h(W_a \cdot a + W_b \cdot b - \theta_{n_1})$. Since $W_a, W_b > 0$, it is easy to verify that

[7] Given $h(x) = 2/(1 + e^{-\beta x}) - 1$, we obtain $h^{-1}(x) = (-1/\beta)\ln((1-x)/(1+x))$. We use the bipolar semi-linear activation function for convenience; any monotonically crescent activation function could have been used here.

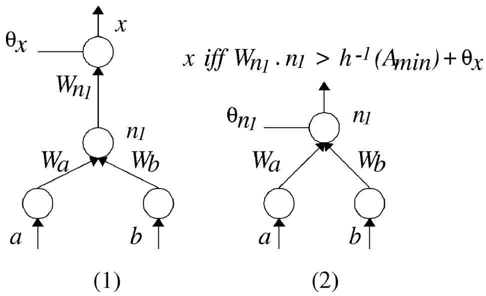


Fig. 4. A single hidden neuron network (1) and its associated constraint (2) with respect to output x. $W_a, W_b, W_{n_1} \in \mathbb{R}^+$.

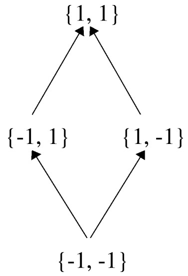


Fig. 5. Ordering on the set of input vectors ($\boldsymbol{I}$) of N.

the ordering of Fig. 5 on the set of input vectors $\boldsymbol{I}$ holds with respect to the output (x) of N. The ordering says, for instance, that the activation of n_1 is maximum if $\boldsymbol{i} = (1, 1)$, that $n_1(1, 1) \geqslant n_1(1, -1)$, and that n_1 is minimum if $\boldsymbol{i} = (-1, -1)$. Since $W_{n_1} > 0$, the activation of x is also maximum if $\boldsymbol{i} = (1, 1)$, and minimum if $\boldsymbol{i} = (-1, -1)$. In other words, the activation value of x is governed by the ordering of Fig. 5.

Given such an ordering, we can draw some conclusions. If the minimum element $(-1, -1)$ is given as the network's input (representing $\sim a \wedge \sim b$), and it activates x, satisfying the constraint $W_{n_1} \cdot n_1 > h^{-1}(A_{min}) + \theta_x$, then any other element in the ordering will also activate x. In this case, since all possible input vectors are in the ordering, we can conclude that x is a fact ($\rightarrow x$). If, on the other hand, the maximum element $(1, 1)$ (representing $a \wedge b$) does not activate x then no other element in the ordering does. As a result, no rule with conclusion x should be obtained from the network. Similarly, if it is the case that both $(1, 1)$ (representing $a \wedge b$) and $(1, -1)$ (representing $a \wedge \sim b$) activate x, that is, $a \wedge b \rightarrow x$ and $a \wedge \sim b \rightarrow x$, then we can conclude that $a \rightarrow x$, regardless of the activation value of b. In this case, the rule $a \rightarrow x$ has been derived as a simplification of the rules $a \wedge b \rightarrow x$ and $a \wedge \sim b \rightarrow x$, which, in turn, have been obtained from (querying) the network. [8]

[8] Throughout, we use the term "to *query* the network" as a short for "to present an input vector to a network and obtain its output vector".

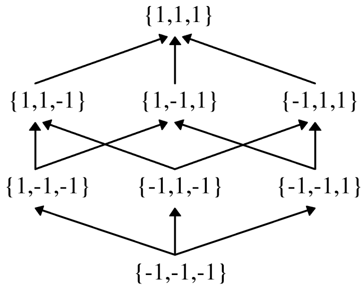


Fig. 6. Partial ordering with respect to set inclusion on the powerset of $\{a, b, c\}$.

We have identified, therefore, that if for all $i, j \in \mathbb{N}$, $W_{ij} \in \mathbb{R}^+$ then it is easy to find an ordering on the set of input vectors ($\boldsymbol{I}$) with respect to the set of output vectors ($\boldsymbol{O}$). Such information can be very useful to guide a pedagogical extraction procedure of symbolic knowledge from the network. The ordering can help reduce the search space, so that we can safely avoid checking irrelevant input vectors, in the sense that those vectors that are not checked would not generate new rules. Moreover, each rule obtained is sound because the extraction is done by querying the actual network.

Notice that in the worst case we still have to check 2^n input vectors, and in the best case we only need to check one input vector (either the minimum or the maximum element in the ordering). Note also that there exists, in fact, a pre-order on the set of input vectors, which, however, may be impossible to find without querying each input vector for a particular set of weights.

Let us now try and see if we can find an ordering easily in the case where there are three inputs $\{a, b, c\}$, but still with $W_{ij} \in \mathbb{R}^+$. It seems reasonable to consider the ordering of Fig. 6 since we do not have any extra information regarding the network's weights. The ordering is built starting from element $(-1, -1, -1)$ and then flipping each input at a time from -1 to 1 until $(1, 1, 1)$ is obtained.

It seems that, for an arbitrary number of input and hidden neurons, if $W_{ij} \in \mathbb{R}^+$, then there exists a unique minimal element $(-1, -1, \ldots, -1)$ and a unique maximum element $(1, 1, \ldots, 1)$ in the ordering on the set of input vectors with respect to the activation values of the output neurons. It seems that $W_{ij} \in \mathbb{R}^+$ is a sufficient condition for the existence of an easily found ordering on the set of input vectors. Let us see if we can confirm this.

We assume the following conventions. Let $\boldsymbol{P}$ be a finite set of literals.[9] Recall that an *interpretation* is a function from $\boldsymbol{P}$ to $\{true, false\}$. Given a neural network N, we associate each input and output neuron with a unique literal in $\boldsymbol{P}$. Let $\mathcal{I}$ be the set of input neurons and $\mathcal{O}$ the set of output neurons of N. Then, each input vector $\boldsymbol{i}$ can be seen as an interpretation, as follows: Suppose $\mathcal{I} = \{p, q, r\}$. We fix a linear ordering on the symbols of $\mathcal{I}$ and represent it as a list, say $[p, q, r]$. We represent $\boldsymbol{i}$ as a string of 1's and -1's, where the value 1 in a particular position in the string means that the literal at the corresponding position in the list of symbols is assigned *true*, and the value -1 means that it is assigned *false*. For example, if $\boldsymbol{i} = (1, -1, 1)$ then $\boldsymbol{i}(p) = \boldsymbol{i}(r) = true$ and $\boldsymbol{i}(q) = false$.

[9] A literal is a propositional variable or the negation of a propositional variable.

Each input vector $\boldsymbol{i}$ can be seen as an abstract representation of a subset of the set of input neurons, with 1's denoting the presence and -1's denoting the absence of a neuron in the set. For example, given the set of input neurons $\mathcal{I}$ as the list $[p, q, r]$, if $\boldsymbol{i} = (1, -1, 1)$ then it represents the set $\{p, r\}$, if $\boldsymbol{i} = (-1, -1, -1)$, it represents $\{\emptyset\}$, if $\boldsymbol{i} = (1, 1, 1)$, it represents $\{p, q, r\}$, and so on. Thus, the set of input vectors $\boldsymbol{I}$ is an abstract representation of the power set of the set of input neurons $\mathcal{I}$. We write it as $\boldsymbol{I} = \wp(\mathcal{I})$.

We are now in a position to formalize the above concepts. We start by defining a distance function between input vectors. The distance between two input vectors is the number of neurons assigned different inputs by each vector. In terms of the above analogy between input vectors and interpretations, the same distance function can be defined as the number of propositional variables with different truth-values.

Definition 8. Let $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ be two input vectors in $\boldsymbol{I}$. The distance $dist(\boldsymbol{i}_m, \boldsymbol{i}_n)$ between $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ is the number of inputs i_j for which $\boldsymbol{i}_m(i_j) \neq \boldsymbol{i}_n(i_j)$, where $\boldsymbol{i}(i_j)$ denotes the input value i_j of vector $\boldsymbol{i}$ ($dist : \boldsymbol{I} \times \boldsymbol{I} \to \mathbb{N}$).

For example, the distance between $\boldsymbol{i}_1 = (-1, -1, 1)$ and $\boldsymbol{i}_2 = (1, 1, -1)$ is $dist(\boldsymbol{i}_1, \boldsymbol{i}_2) = 3$. The distance between $\boldsymbol{i}_3 = (-1, 1, -1)$ and $\boldsymbol{i}_4 = (1, -1, -1)$ is $dist(\boldsymbol{i}_3, \boldsymbol{i}_4) = 2$.

Another concept that will prove to be important is the sum of the input elements in a input vector. We define it as follows.

Definition 9. Let $\boldsymbol{i}_m$ be a p-ary input vector in $\boldsymbol{I}$. The sum $\langle \boldsymbol{i}_m \rangle$ of $\boldsymbol{i}_m$ is the sum of all input elements i_j in $\boldsymbol{i}_m$, that is $\langle \boldsymbol{i}_m \rangle = \sum_{j=1}^{p} \boldsymbol{i}_m(i_j)$ ($\langle\,\rangle : \boldsymbol{I} \to \mathbb{Z}$).

For example, the sum of $\boldsymbol{i}_1 = (-1, -1, 1)$ is $\langle \boldsymbol{i}_1 \rangle = -1$. The sum of $\boldsymbol{i}_2 = (1, 1, -1)$ is $\langle \boldsymbol{i}_2 \rangle = 1$.

Now we define the ordering $\leqslant_{\boldsymbol{I}}$ on $\boldsymbol{I} = \wp(\mathcal{I})$ with respect to set inclusion. Recall that $\boldsymbol{i}_m \in \boldsymbol{I}$ is an abstract representation of a subset of $\mathcal{I}$. We say that $\boldsymbol{i}_m \subseteq \boldsymbol{i}_n$ if the set represented by $\boldsymbol{i}_m$ is a subset of the set represented by $\boldsymbol{i}_n$.

Definition 10. Let $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ be input vectors in $\boldsymbol{I}$. $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}} \boldsymbol{i}_n$ *iff* $\boldsymbol{i}_m \subseteq \boldsymbol{i}_n$.

Clearly, for a finite set $\mathcal{I}$, $\boldsymbol{I}$ is a finite partially ordered set with respect to $\leqslant_{\boldsymbol{I}}$ having $\mathcal{I}$ as its maximum element and the empty set $\emptyset$ as its minimum element. In other words, $\sup(\boldsymbol{I}) = \{1, 1, \ldots, 1\}$ and $\inf(\boldsymbol{I}) = \{-1, -1, \ldots, -1\}$.

The following Proposition 11 shows that $\leqslant_{\boldsymbol{I}}$ is actually an ordering of interest with respect to the network's output.

Proposition 11. *If* $W_{ji} \in \mathbb{R}^+$ *then* $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}} \boldsymbol{i}_n$ *implies* $o_j(\boldsymbol{i}_m) \leqslant o_j(\boldsymbol{i}_n)$, *for all* $1 \leqslant j \leqslant q$.

Proof. Let $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}} \boldsymbol{i}_n$ and $dist(\boldsymbol{i}_m, \boldsymbol{i}_n) = 1$, then $\boldsymbol{i}_m(i_i) = -1$ and $\boldsymbol{i}_n(i_i) = 1$ for some input i_i. Let r be the number of hidden neurons in the network. Firstly, we have to show that:

$$
\begin{aligned}
&h\left(\sum_{i=1}^{p}\left(W_{1i}^{1}\boldsymbol{i}_m(i_i)-\theta_{n_1}\right)\right)+h\left(\sum_{i=1}^{p}\left(W_{2i}^{1}\boldsymbol{i}_m(i_i)-\theta_{n_2}\right)\right)+\cdots\\
&\quad+h\left(\sum_{i=1}^{p}\left(W_{ri}^{1}\boldsymbol{i}_m(i_i)-\theta_{n_r}\right)\right)\\
&\qquad\leqslant h\left(\sum_{i=1}^{p}\left(W_{1i}^{1}\boldsymbol{i}_n(i_i)-\theta_{n_1}\right)\right)\\
&\qquad\quad+h\left(\sum_{i=1}^{p}\left(W_{2i}^{1}\boldsymbol{i}_n(i_i)-\theta_{n_2}\right)\right)+\cdots+h\left(\sum_{i=1}^{p}\left(W_{ri}^{1}\boldsymbol{i}_n(i_i)-\theta_{n_r}\right)\right).
\end{aligned}
$$

By the definition of $\leqslant_{\boldsymbol{I}}$ and since $W_{ji}\in\mathbb{R}^{+}$ we derive immediately that for all j $(1\leqslant j\leqslant r)$

$$
\sum_{i=1}^{p}\left(W_{ji}^{1}\boldsymbol{i}_m(i_i)-\theta_{n_j}\right)\leqslant\sum_{i=1}^{p}\left(W_{ji}^{1}\boldsymbol{i}_n(i_i)-\theta_{n_j}\right),
$$

and by the monotonically crescent characteristic of $h(x)$ we obtain $\forall j$ $(1\leqslant j\leqslant r)$ $h(\sum_{i=1}^{p}(W_{ji}^{1}\boldsymbol{i}_m(i_i)-\theta_{n_j}))\leqslant h(\sum_{i=1}^{p}(W_{ji}^{1}\boldsymbol{i}_n(i_i)-\theta_{n_j}))$. This proves that if $\boldsymbol{i}_m\leqslant_{\boldsymbol{I}}\boldsymbol{i}_n$ and $\mathit{dist}(\boldsymbol{i}_m,\boldsymbol{i}_n)=1$ then $n_j(\boldsymbol{i}_m)\leqslant n_j(\boldsymbol{i}_n)$ for all $1\leqslant j\leqslant r$. In the same way, we obtain that $h(\sum_{i=1}^{r}(W_{ji}^{2}\boldsymbol{n}_m(n_i)-\theta_{o_j}))\leqslant h(\sum_{i=1}^{r}(W_{ji}^{2}\boldsymbol{n}_n(n_i)-\theta_{o_j}))$, and, therefore, that:

$$
\text{if } \mathit{dist}(\boldsymbol{i}_m,\boldsymbol{i}_n)=1 \text{ then } o_j(\boldsymbol{i}_m)\leqslant o_j(\boldsymbol{i}_n) \text{ for } 1\leqslant j\leqslant q. \tag{4}
$$

Now, let $\boldsymbol{i}_m\leqslant_{\boldsymbol{I}}\boldsymbol{i}_n$ and $\mathit{dist}(\boldsymbol{i}_m,\boldsymbol{i}_n)=k$ $(1<k\leqslant p)$. There are $k-1$ vectors $\boldsymbol{i}_\xi,\ldots,\boldsymbol{i}_\zeta$ such that $\boldsymbol{i}_m\leqslant_{\boldsymbol{I}}\boldsymbol{i}_\xi\leqslant_{\boldsymbol{I}}\cdots\leqslant_{\boldsymbol{I}}\boldsymbol{i}_\zeta\leqslant_{\boldsymbol{I}}\boldsymbol{i}_n$. From (4) above and since $\leqslant$ is transitive, it follows that if $\boldsymbol{i}_m\leqslant_{\boldsymbol{I}}\boldsymbol{i}_n$ then $o_j(\boldsymbol{i}_m)\leqslant o_j(\boldsymbol{i}_n)$ for all $1\leqslant j\leqslant q$. □

4.2. Regular networks

Let us see now if we can relax the condition $W_{ji}\in\mathbb{R}^{+}$ and still find easily an ordering on the set of input vectors of a network. We start by giving an example.

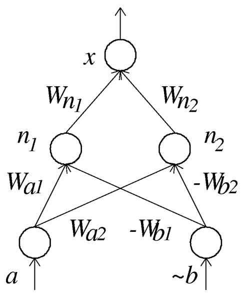


Fig. 7. The positive form of a (regular) network.

Example 12. Consider the network N of Fig. 2. Assume W_{b1} and $W_{b2} < 0$. Although some weights are negative, we can find a "regularity" in the network. For example, input neuron b contributes negatively to the activation of both n_1 and n_2, and there are no negative connections from the hidden to the output layer of N. Following [10], we can transform the network of Fig. 2 into the network of Fig. 7, where all weights are positive and input neuron b is negated.

Given the network of Fig. 7, we can find an ordering on the set of input vectors in the same way as before. The only difference is that now $\mathcal{I} = \{a, \sim b\}$. We will see later that, if we account for the fact that $\mathcal{I}$ may now have negated literals (default negation), then the networks of Figs. 2 and 7 are equivalent.

Let us analyze what we have done in the above example. We continue to assume that the weights from the hidden layer to any one neuron in the output layer of a network are either all positive or all negative. Then, for each input neuron y, we do the following:

(1) If y is linked to the hidden layer through connections with positive weights only:
   (a) do nothing.
(2) If y is linked to the hidden layer through connections with negative weights W_{jy} only:
   (a) change each W_{jy} to $-W_{jy}$ and rename y by $\sim y$.
(3) If y is linked to the hidden layer through positive and negative connections:
   (a) add a neuron named $\sim y$ to the input layer, and
   (b) for each negative connection with weight W_{jy} from y to n_j:
      (i) add a new connection with weight $-W_{jy}$ from $\sim y$ to n_j, and
      (ii) delete the connection with weight W_{jy} from y to n_j.

We call the above procedure the *Transformation Algorithm*.

Example 13. Consider again the network of Fig. 2, but now assume that only $W_{a2} < 0$. Applying the Transformation Algorithm, we obtain the network of Fig. 8.

Although the network of Fig. 8 has positive weights only, it is clearly not equivalent to the original network of Fig. 2. In this case, the combination of n_1 and n_2 is not straightforward. Note that, $\boldsymbol{i} = (1, 1)$ in the original network provides the maximum activation of n_1, but not the maximum activation of n_2; that is given by $\boldsymbol{i} = (-1, 1)$. We can

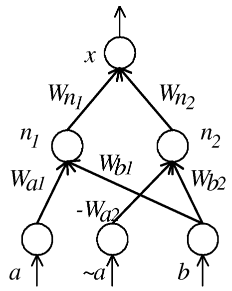


Fig. 8. The positive form of a (non-regular) network.

not affirm anymore that $(1,1)$ is bigger than $(-1,1)$ with respect to the output x, without having to check them by querying the network.

Examples 12 and 13 indicate that if the Transformation Algorithm generates a network where complementary literals (say, a and $\sim a$) appear in the input layer (see the network of Fig. 8) then the ordering $\leqslant_{\boldsymbol{I}}$ on $\boldsymbol{I}$ is not applicable. However, if complementary literals do not appear in the input layer of the network obtained from the above transformation (see Fig. 7), it seems that $\leqslant_{\boldsymbol{I}}$ is still valid for such networks, which have "well-behaved" negative weights. This motivates the following definition.

Definition 14. A single hidden layer neural network is said to be *regular* if its connections from the hidden layer to each output neuron have either all positive or all negative weights, and if the above Transformation Algorithm generates on it a network without complementary literals in the input layer.

Returning to Example 12, we have seen that the positive form N_+ of a regular network N may have negated literals in the set of input neurons (e.g., $\mathcal{I}_+ = \{a, \sim b\}$). In this case, if we represent $\mathcal{I}_+$ as a list, say $[a, \sim b]$, and refer to an input vector $\boldsymbol{i} = (-1, 1)$ with respect to $\mathcal{I}_+$, then we consider $\boldsymbol{i}$ as the abstract representation of the set $\{\sim b\}$. In the same way, $\boldsymbol{i} = (1, -1)$ represents $\{a\}$, and so on. In this sense, the set of input vectors of N_+ can be ordered with respect to set inclusion exactly as before, using Definition 10, as the following example illustrates.

Example 15. Consider the network N_+ of Fig. 7. Given $\mathcal{I}_+ = [a, \ \sim b]$, we obtain the ordering of Fig. 9(1) with respect to set inclusion. The ordering of Fig. 9(2) on the set of input vectors of the original network N is obtained by mapping each element of (1) into (2) using $\sim b = 1$ implies $b = -1$, and $\sim b = -1$ implies $b = 1$. As a result, querying N_+ with $\boldsymbol{i} = (1, 1)$ is equivalent to querying N with $\boldsymbol{i} = (1, -1)$, querying N_+ with $\boldsymbol{i} = (-1, 1)$ is equivalent to querying N with $\boldsymbol{i} = (-1, -1)$, and so on.

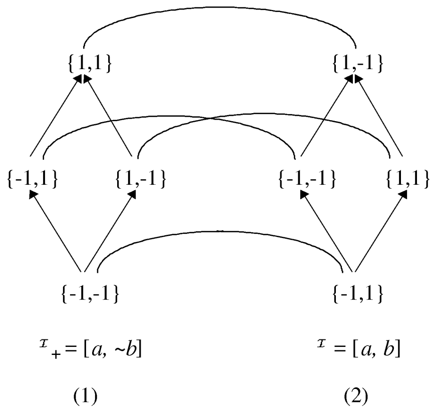


Fig. 9. The ordering with respect to set inclusion on the positive form of a network (1) and the ordering on the original network (2).

More precisely, we define the function σ mapping input vectors of the positive form into input vectors of the original network, as follows. Let $\boldsymbol{I}$ be the set of input vectors of s tuples. Given the set of input neurons $\mathcal{I}_+$ and an abstract representation $\boldsymbol{I}_+$ of $\wp(\mathcal{I}_+)$, each element $x_i \in \mathcal{I}_+$, $1 \leqslant i \leqslant s$, is mapped to the set $\{-1, 1\}$ such that $\sigma_{[x_1,\ldots,x_s]}(i_1, \ldots, i_s) = (i'_1, \ldots, i'_s)$, where $i'_i = i_i$ if x_i is a positive literal and $i'_i = -i_i$ if x_i is a negative literal. For example $\sigma_{[a,\sim b,c,\sim d]}(1, 1, -1, -1) = (1, -1, -1, 1)$.

Note that the correspondence between input vectors and interpretations is still valid. We only need to define $\boldsymbol{i}(\sim p) = \mathit{false}$ iff $\boldsymbol{i}(p) = \mathit{true}$ and $\sim\sim p = p$. For example, for $\mathcal{I}_+ = [a, \sim b]$, if $\boldsymbol{i} = (-1, -1)$ then $\boldsymbol{i}(a) = \mathit{false}$ and $\boldsymbol{i}(b) = \mathit{true}$.

Proposition 16. *Let $\mathcal{I}_+$ be the set of input neurons of the positive form N_+ of a regular network N. Let $\boldsymbol{I}_+ = \wp(\mathcal{I}_+)$ be ordered under the set inclusion relation $\leqslant_{\boldsymbol{I}_+}$, and $\boldsymbol{i}_m, \boldsymbol{i}_n \in \boldsymbol{I}_+$. Thus, $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}_+} \boldsymbol{i}_n$ implies $o_j(\sigma_{[\mathcal{I}_+]}(\boldsymbol{i}_m)) \leqslant o_j(\sigma_{[\mathcal{I}_+]}(\boldsymbol{i}_n))$, for all $1 \leqslant j \leqslant q$ in N.*

Proof. Straightforward by Proposition 11 and by the above definition of the mapping function σ. □

Proposition 16 establishes the correlation between regular networks and their positive counterpart. As a result, the extraction procedure can either use the set inclusion ordering on $\mathcal{I}_+$ (as, e.g., in Fig. 9(1)), and query directly the positive form of the network, or use the mapping function σ to obtain the ordering on the regular, original network (Fig. 9(2)), and query the original network. We will adopt the first policy. Note that if the network is already positive then σ is the identity function.

We have seen briefly that if we can find an ordering on the set of input vectors of a network, there are some properties that can help reducing the search space of input vectors during a pedagogical extraction of rules. Let us now define precisely these properties.

Proposition 17 (Search Space Pruning Rule 1). *Let $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ be input vectors of the positive form of a regular neural network N such that dist$(\boldsymbol{i}_m, \boldsymbol{i}_n) = 1$ and $\langle \boldsymbol{i}_m \rangle < \langle \boldsymbol{i}_n \rangle$. If $\boldsymbol{i}_n$ does not satisfy the constraint $\boldsymbol{C}_{o_j}$ on the jth output neuron of N, then $\boldsymbol{i}_m$ does not satisfy $\boldsymbol{C}_{o_j}$ either.*

Proof. Directly by Definitions 8, 9 and 10, if dist$(\boldsymbol{i}_m, \boldsymbol{i}_n) = 1$ and $\langle \boldsymbol{i}_m \rangle < \langle \boldsymbol{i}_n \rangle$ then $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}} \boldsymbol{i}_n$. By Proposition 11, $o_j(\boldsymbol{i}_m) \leqslant o_j(\boldsymbol{i}_n)$. That completes the proof. □

Proposition 18 (Search Space Pruning Rule 2). *Let $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ be input vectors of the positive form of a regular neural network N, such that dist$(\boldsymbol{i}_m, \boldsymbol{i}_n) = 1$ and $\langle \boldsymbol{i}_m \rangle < \langle \boldsymbol{i}_n \rangle$. If $\boldsymbol{i}_m$ satisfies the constraint $\boldsymbol{C}_{o_j}$ on the jth output neuron of N, then $\boldsymbol{i}_n$ also satisfies $\boldsymbol{C}_{o_j}$.*

Proof. This is the contrapositive of Proposition 17. □

Proposition 17 says that for any $\boldsymbol{i} \in \boldsymbol{I}$, starting from $\sup(\boldsymbol{I})$, if $\boldsymbol{i}$ does not activate the jth output neuron o_j, then the immediate predecessors of $\boldsymbol{i}$ do not activate o_j either. Similarly, Proposition 18 says that for any $\boldsymbol{i} \in \boldsymbol{I}$, starting from $\inf(\boldsymbol{I})$, if $\boldsymbol{i}$ does activate the jth output neuron o_j, then the immediate successors of $\boldsymbol{i}$ also do.

In Example 7, we have seen briefly that the extracted rules $ab \to x$ and $a{\sim}b \to x$ could be simplified to obtain a single rule, namely, $a \to x$. Let us now define a group of *simplification rules* that will help in the extraction of a smaller and clearer set of rules. They will also help reducing the number of premises per rule, an important aspect of readability.

Definition 19 (*Subsumption*). A rule r_1 subsumes a rule r_2 iff r_1 and r_2 have the same conclusion and the set of premises of r_1 is a subset of the set of premises of r_2.

For example, $a \to x$ subsumes $ab \to x$ and $a{\sim}b \to x$.

Definition 20 (*Complementary literals*). Let $r_1 = L_1, \ldots, L_i, \ldots, L_j \to L_{j+1}$ and $r_2 = L_1, \ldots, {\sim}L_i, \ldots, L_j \to L_{j+1}$ be extracted rules, where $j \leqslant |\mathcal{I}|$. Then, $r_3 = L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_j \to L_{j+1}$ is also an extracted rule. Note that r_3 subsumes r_1 and r_2.

For example, if $\mathcal{I} = \{a, b, c\}$ and we write $a{\sim}b \to x$, then it simplifies $a{\sim}bc \to x$ and $a{\sim}b{\sim}c \to x$. Note that, considering the ordering on $\boldsymbol{I}$, the above property requires that two adjacent input vectors, $\boldsymbol{i}_m = (1, -1, 1)$ and $\boldsymbol{i}_n = (1, -1, -1)$, activate x.

Definition 21 (*Fact*). If a literal L_{j+1} holds in the presence of any combination of the truth values of literals $L_1, \ldots, L_j$ in $\mathcal{I}$ then we derive a rule of the form $\to L_{j+1}$ (L_{j+1} is a fact).

Definition 21 is an important special case of Definition 20. Considering the ordering on $\boldsymbol{I}$, an output neuron x is a fact iff $\inf(\boldsymbol{I})$ activates x. Note that, by Proposition 18, if $\inf(\boldsymbol{I})$ activates x then any other input vector in $\boldsymbol{I}$ also does.

Another interesting special case occurs when $\sup(\boldsymbol{I})$ does not activate x. In this case, by Proposition 17, no other input vector in $\boldsymbol{I}$ activates x, and, thus, there are no rules with conclusion x to be derived from the network.

Definition 22 (*M of N*). Let $m, n \in \mathbb{N}, \mathcal{I}' \subseteq \mathcal{I}, |\mathcal{I}'| = n, m \leqslant n$. Then, if any combination of m elements chosen from $\mathcal{I}'$ implies L_{j+1}, we derive a rule of the form $m(\mathcal{I}') \to L_{j+1}$.

The above Definition 22 may be very useful in helping to reduce the number of rules extracted. It states that, for example, $2(abc) \to x$ represents $ab \to x$, $ac \to x$, and $bc \to x$. In this way, if for example we write $3(abcdef) \to x$ then this rule is a short representation of at least $\mathcal{C}_3^6 = 20$ rules. [10]

There is a rather intricate relation between each rule of the form *M of N* and the ordering on the set of input vectors $\boldsymbol{I}$, in the sense that each valid *M of N* rule represents a subset of $\boldsymbol{I}$. Here is a flavor of that relation in an example where it is easy to identify it. Suppose

[10] Note that if $\mathcal{I} = \{a, b, c\}$ and we write $1(ab) \to x$, then such an *M of N* rule is a simplification of $\mathcal{C}_1^2 = 2$ rules: $a \to x$ and $b \to x$. However, by Definition 20, $a \to x$ and $b \to x$ are already simplifications of $abc \to x$, $ab{\sim}c \to x$, $a{\sim}bc \to x$, $a{\sim}b{\sim}c \to x$, ${\sim}abc \to x$, *and* ${\sim}ab{\sim}c \to x$.

$\mathcal{I} = \{a, b, c\}$ and assume that $\mathcal{I}' = \mathcal{I}$. Let us say that the output neuron in question is x and that constraint $\boldsymbol{C}_{o_x}$ is satisfied by at least one input vector in $\boldsymbol{I}$. If only $\sup(\boldsymbol{I})$ satisfies $\boldsymbol{C}_{o_x}$, we derive the rule $abc \to x$. Clearly, this rule is equivalent to $3(abc) \to x$. If all the immediate predecessors of $\sup(\boldsymbol{I})$ also satisfy $\boldsymbol{C}_{o_x}$, it is not difficult to verify that the four rules obtained ($r_1 : abc \to x$, $r_2 : ab{\sim}c \to x$, $r_3 : a{\sim}bc \to x$, $r_4 :\ {\sim}abc \to x$) can be represented by $2(abc) \to x$. This is because, by Definition 20, each rule r_2, r_3 and r_4 can be simplified together with r_1, deriving $abc \to x$, $ab \to x$, $ac \to x$ and $bc \to x$. Since, by Definition 19, $abc \to x$ is subsumed by any of the other three rules, we obtain $2(abc) \to x$. Moreover, $2(abc) \to x$ subsumes $3(abc) \to x$. This motivates the definition of yet another simplification rule, as follows.

Definition 23 (*M of N subsumption*). Let $m, p \in \mathbb{N}, \mathcal{I}' \subseteq \mathcal{I}$. $m(\mathcal{I}') \to L_{j+1}$ subsumes $p(\mathcal{I}') \to L_{j+1}$ iff $m < p$.

Returning to the illustration about the relation between M of N rules and subsets of $\boldsymbol{I}$, let us see what happens if the elements at distance 2 from $\sup(\boldsymbol{I})$ all satisfy $\boldsymbol{C}_{o_x}$. We expect that the set of rules obtained from $\boldsymbol{I}$ could be represented by $1(abc) \to x$, and in fact it is. From the elements at distance 2 from $\sup(\boldsymbol{I})$, we obtain the following rules: $r_1 : a{\sim}b{\sim}c \to x$, $r_2 :\ {\sim}ab{\sim}c \to x$, and $r_3 :\ {\sim}a{\sim}bc \to x$. By Proposition 18, we know that the elements at distance 1 from $\sup(\boldsymbol{I})$ also satisfy $\boldsymbol{C}_{o_x}$, and we derive the rules: $r_4 : ab{\sim}c \to x$, $r_5 : a{\sim}bc \to x$, and $r_6 :\ {\sim}abc \to x$. Again by Proposition 18, $\sup(\boldsymbol{I})$ itself also satisfies $\boldsymbol{C}_{o_x}$, and we derive $r_7 : abc \to x$. Now, applying Definition 20 over r_1 and r_4, we obtain the simplified rule $r_8 : a{\sim}c \to x$, taking r_5 and r_7, we obtain $r_9 : ac \to x$, and from r_8 and r_9, we derive $r_a : a \to x$. Similarly, from r_2, r_4, r_6 and r_7, we derive $r_b : b \to x$, and from r_3, r_5, r_6 and r_7, we derive $r_c : c \to x$. Finally, since r_a, r_b and r_c together subsume any rule previously obtained, by Definition 22 we may derive the single *M of N* rule $1(abc) \to x$.

We have identified a pattern in the ordering on $\boldsymbol{I}$ with respect to a group of *M of N* rules, the ones where $\mathcal{I}' = \mathcal{I}$. More generally, given $|\mathcal{I}| = k$, if all the elements in $\boldsymbol{I}$ that are at distance d from $\sup(\boldsymbol{I})$ satisfy a constraint $\boldsymbol{C}_{o_x}$, then derive the rule $(k-d)(\mathcal{I}) \to x$. Note that there are $\mathcal{C}^k_{k-d}$ elements at distance d from $\sup(\boldsymbol{I})$, and that, as a result of Proposition 18, if all the elements in $\boldsymbol{I}$ at distance d from $\sup(\boldsymbol{I})$ satisfy $\boldsymbol{C}_{o_x}$ then any other element at distance d' from $\sup(\boldsymbol{I})$ such that $0 \leqslant d' < d$ also satisfies $\boldsymbol{C}_{o_x}$.

Remark 1. We have defined regular networks (see Definition 14) either with all the weights from the hidden layer to each output neuron positive or with all of them negative. We have, although, considered in the above examples and definitions only the ones where all the weights are positive. However, it is not difficult to verify that the constraint $\boldsymbol{C}_{o_j}$ on the jth output of a regular network with negative weights from hidden to output layer is $W^2_{j1}n_1 + W^2_{j2}n_2 + \cdots + W^2_{jr}n_r < h^{-1}(A_{min}) + \theta_{o_j}$. As a result, the only difference now is on the sign ($<$) of the constraint. In other words, in this case we only need to invert the signs at Propositions 17 and 18. All remaining definitions and propositions are still valid.

We have so far referred to soundness and completeness of the extraction algorithm in a somewhat vague manner. Let us define these concepts precisely.

Definition 24 (*Extraction algorithm soundness*). A rules' extraction algorithm from a neural network N is *sound* iff for each rule r_i extracted, whenever the premise of r_i is presented to N as input vector, in the presence of any combination of the input values of literals not referenced by rule r_i, the conclusion of r_i presents activation greater than A_{min} in the output vector of N.

Definition 25 (*Extraction algorithm completeness*). A rules' extraction algorithm from a neural network N is *complete* iff each rule extracted by exhaustively verifying all the combinations of the input vector of N either belongs to, or is subsumed by, a rule in the set of rules generated by the extraction algorithm.

We are finally in a position to present the extraction algorithm for regular networks, which will be refined in Section 5 for the general case extraction.

Knowledge extraction algorithm for regular networks [11]

(1) Apply the *Transformation Algorithm* over N, obtaining its positive form N_+;
(2) Find $\inf(\boldsymbol{I})$ and $\sup(\boldsymbol{I})$ with respect to N_+ using σ;
(3) For each neuron o_j in the output layer of N_+ do:
  (a) Query N_+ with input vector $\inf(\boldsymbol{I})$. If $o_j > A_{min}$, apply the Simplification Rule *Fact* and stop.
  (b) Query N_+ with input vector $\sup(\boldsymbol{I})$. If $o_j \leqslant A_{min}$, stop.

  /∗ *Search the input vectors' space* $\boldsymbol{I}$.

  (c) $\boldsymbol{i}_\perp := \inf(\boldsymbol{I})$; $\boldsymbol{i}_\top := \sup(\boldsymbol{I})$;
  (d) While $dist(\boldsymbol{i}_\perp, \inf(\boldsymbol{I})) \leqslant n\,\mathrm{DIV}\,2$ or $dist(\boldsymbol{i}_\top, \sup(\boldsymbol{I})) \leqslant n\,\mathrm{DIV}\,2 + n\,\mathrm{MOD}\,2$, where n is the number of input neurons of N_+, and still generating new $\boldsymbol{i}_\perp$ or $\boldsymbol{i}_\top$, do:

  /∗ *Generate the successors of* $\boldsymbol{i}_\perp$ *and query the network*

    (i) set new $\boldsymbol{i}_\perp :=$ old $\boldsymbol{i}_\perp$ flipped according to the ordering on $\boldsymbol{I}$; [12]
    (ii) Query N_+ with input vector $\boldsymbol{i}_\perp$;
    (iii) If Search Space *Pruning Rule 2* is applicable, stop generating new $\boldsymbol{i}_\perp$;
    (iv) Apply the Simplification Rule *Complementary Literals*, and Add the rules derived accordingly to the rule set.

    /∗ *Generate the predecessors of* $\boldsymbol{i}_\top$ *and query the network*

    (v) set new $\boldsymbol{i}_\top :=$ old $\boldsymbol{i}_\top$ flipped according to the ordering on $\boldsymbol{I}$; [13]
    (vi) Query N_+ with input vector $\boldsymbol{i}_\top$;
    (vii) If Search Space *Pruning Rule 1* is applicable, stop generating new $\boldsymbol{i}_\top$;

[11] The algorithm is kept simple for clarity, and is not necessarily the most efficient.
[12] From $\inf(\boldsymbol{I})$, we generate new $\boldsymbol{i}_\perp$ by flipping the elements at old $\boldsymbol{i}_\perp$ from right to left.
[13] From $\sup(\boldsymbol{I})$, we generate new $\boldsymbol{i}_\top$ by flipping the elements at old $\boldsymbol{i}_\top$ from left to right.

(viii) Apply the Simplification Rule *M of N*, and Add the rules derived accordingly to the rule set.

(e) Apply the Simplification Rules *Subsumption* and *M of N Subsumption* on the rule set regarding o_j.

Note that if the weights from the hidden to the output layer of N are negative, we simply substitute $\inf(\boldsymbol{I})$ by $\sup(\boldsymbol{I})$ and vice-versa. In a given application, the above extraction algorithm can be halted if a desired degree of accuracy is achieved in the set of rules. The algorithm is such that the exact symbolic representation of the network is being approximated at each cycle.

Example 26. Suppose $\mathcal{I} = \{a, b, c\}$ and let $\boldsymbol{I} = \wp(\mathcal{I})$ be ordered with respect to set inclusion. We start by checking $\inf(\boldsymbol{I})$ with respect to an output neuron x. If $\inf(\boldsymbol{I})$ activates x, i.e., $\inf(\boldsymbol{I})$ satisfies constraint C_{o_x}, then by Proposition 18 any other input vector activates x and by Definition 21 we can extract $\rightarrow x$ and stop. If, on the other hand, $\inf(\boldsymbol{I})$ does not activate x, then we may need to query the network with the immediate successors of $\inf(\boldsymbol{I})$. Let us call these input vectors $\boldsymbol{I}^*$, where $dist(\inf(\boldsymbol{I}), \boldsymbol{I}^*) = 1$.

We proceed to check the element $\sup(\boldsymbol{I})$. If $\sup(\boldsymbol{I})$ does not satisfy C_{o_x}, by Proposition 17 we can stop, extracting no rules with conclusion x. If $\sup(\boldsymbol{I})$ activates x, we conclude that $abc \rightarrow x$, but we still have to check the input vectors $\boldsymbol{I}^{**}$ at distance 1 from $\sup(\boldsymbol{I})$. We may also later apply some simplification on $abc \rightarrow x$, if at least one of the input vectors in $\boldsymbol{I}^{**}$ activates x. Hence, we keep $abc \rightarrow x$ in stand by and proceed.

Let us say that we choose to start by checking $\boldsymbol{i}_1 = (-1, -1, 1)$ in $\boldsymbol{I}^*$. If $\boldsymbol{i}_1$ does not satisfy C_{o_x}, we have to check the remaining inputs in $\boldsymbol{I}^*$. However, if $\boldsymbol{i}_1$ activates x then, again by Proposition 18, we know that $(-1, 1, 1)$ and $(1, -1, 1)$ also do. This tells us that not all the inputs in $\boldsymbol{I}^{**}$ need to be checked. Moreover, if all the elements in $\boldsymbol{I}^*$ activate x then we can use Definition 22 to derive $1(abc) \rightarrow x$ and stop the search.

Analogously, when checking $\boldsymbol{I}^{**}$ we can obtain information about $\boldsymbol{I}^*$. If, for instance, $\boldsymbol{i}_2 = (1, 1, -1)$ does not activate x then $(-1, 1, -1)$ and $(1, -1, -1)$ in $\boldsymbol{I}^*$ do not either, now by Proposition 17. If, on the contrary, $\boldsymbol{i}_2$ activates x, we can derive $ab \rightarrow x$, using Proposition 18 and Definition 20. If not only $\boldsymbol{i}_2$ but also the other inputs in $\boldsymbol{I}^{**}$ activate x then we obtain $2(abc) \rightarrow x$, which subsumes $abc \rightarrow x$ by Definitions 22 and 19. In this case, we still need to query the network with inputs $\boldsymbol{i}$ at distance 1 from $\boldsymbol{i}_2$ such that $\langle \boldsymbol{i} \rangle < \langle \boldsymbol{i}_2 \rangle$, but those inputs are already the ones in $\boldsymbol{I}^{**}$ and therefore we can stop. Note that the stopping criteria are the following: either all elements in the ordering are visited or, if not, for each element not visited, Propositions 17 and 18 guarantee that it is safe not to consider it, in the sense that it is either already represented in the set of rules, or irrelevant and cannot give rise to any new rule.

Theorem 27 (Soundness). *The extraction algorithm for regular networks is sound* (*satisfies Definition* 24).

Proof. We have to show that, whether a rule r is extracted by querying the network (Case 1) or by a simplification of rules (Case 2), any rule r' that is subsumed by r, including r itself, can be obtained by querying the network. We prove this by contradiction. Consider a set $\boldsymbol{I}$ of p-ary input vectors. Assume that there exist rules r and r' such that r' is subsumed by r, and r' is not obtainable by querying the network. Assume also that r contains the largest number of premises of such a rule. Let X_i denote L_i *or* $\sim L_i$ $(1 \leqslant i \leqslant p)$.

Case 1: If r is itself obtained by querying the network, then the only possible subsumed rule is r, and obviously this yields a contradiction.

Case 2: r is either a simplification by *Complementary Literals*, or a *Fact*, or a *M of N* rule. It is shown that each assumption yields a contradiction.

Let $r = L_1, \ldots, L_q \rightarrow L_j$ $(1 \leqslant q < p)$ be a simplification by *Complementary Literals*. Then, r is derived from two rules $r_1' = L_1, \ldots, L_s, \ldots, L_q \rightarrow L_j$ and $r_2' = L_1, \ldots, {\sim}L_s, \ldots, L_q \rightarrow L_j$ $(1 \leqslant s \leqslant q)$. Each of these has more premises than r. So, by assumption, all rules subsumed by r_1' and r_2' are obtainable by querying the network. By Proposition 18, r is also obtained by querying the network. Since, by Definition 19, any other rule subsumed by r is also subsumed by either r_1' or by r_2', this leads to a contradiction.

Let $r = \rightarrow L_j$ be a simplification by *Fact*. Then, r must have been obtained by querying the network with $\inf(\boldsymbol{I})$. By Proposition 18, any rule of the form $X_1, \ldots, X_p \rightarrow L_j$ is also obtainable by querying the network, contradicting the assumption about r'.

Finally, if a further simplification is made, to obtain $r = m(L_1, \ldots, L_n) \rightarrow L_j$ $(1 \leqslant m < n \leqslant p)$ by *M of N* simplification, then r is obtained from a set of rules of the form $L_1, \ldots, L_m \rightarrow L_j$, where $L_1, \ldots, L_m$ are m elements chosen from $\{L_1, \ldots, L_n\}$. By the previous cases, all subsumed rules are obtainable by querying the network. □

Theorem 28 (Completeness). *The extraction algorithm for regular networks is complete* (*satisfies Definition* 25).

Proof. We have to show that the extraction algorithm terminates either when all possible combinations of the input vector have been queried in the network (Case 1) or the set of rules extracted subsumes any rule that would be derived from an element not queried (Case 2). Case 1 is trivial. In Case 2, we have to show that any element not queried either would not generate a rule (Case 2(i)) or would generate a rule that is subsumed by some rule extracted (Case 2(ii)).

Consider a set $\boldsymbol{I}$ of p-ary input vectors.

Case 2(i): Let $\boldsymbol{i}_m, \boldsymbol{i}_n \in \boldsymbol{I}$, $dist(\boldsymbol{i}_m, \boldsymbol{i}_n) = q$ $(1 \leqslant q \leqslant p)$ and $\langle \boldsymbol{i}_m \rangle < \langle \boldsymbol{i}_n \rangle$. Assume that $\boldsymbol{i}_n$ is queried in the network and that $\boldsymbol{i}_n$ does not generate a rule. By Proposition 17 q times, $\boldsymbol{i}_m$ would not generate a rule either.

Case 2(ii): Let $\boldsymbol{i}_k, \boldsymbol{i}_o \in \boldsymbol{I}$, $dist(\boldsymbol{i}_k, \boldsymbol{i}_o) = t$ $(1 \leqslant t \leqslant p)$ and $\langle \boldsymbol{i}_k \rangle < \langle \boldsymbol{i}_o \rangle$. Assume that $\boldsymbol{i}_k$ is queried in the network and that $\boldsymbol{i}_k$ derives a rule r_k. Let $S = \{L_1, \ldots, L_s\}$ be the set of positive literals in the body of r_k, where $s \in [1, p]$. By Definition 20, the rule $r = L_1, \ldots, L_s \rightarrow L_j$ can be obtained from r_k. Clearly, r subsumes r_k. Now, by Proposition 18 t times, $\boldsymbol{i}_o$ would also derive a rule r_o. Let $U = \{L_1, \ldots, L_u\}$ be the set of positive literals in the body of r_o, where $u \in [1, p]$. Since $\langle \boldsymbol{i}_k \rangle < \langle \boldsymbol{i}_o \rangle$ then $S \subset U$ and, by Definition 19, r also subsumes r_o.

That completes the proof since all the stopping criteria of the extraction algorithm have been covered. □

5. The extraction algorithm for non-regular networks

So far, we have seen that for the case of regular networks it is possible to apply an ordering on the set of input vectors, and use a sound and complete pedagogical extraction algorithm that searches for relevant input vectors in this ordering. Furthermore, the neural network and its set of rules can be shown equivalent (that results directly from the proofs of soundness and completeness of the extraction algorithm).

Despite the above results being highly desirable, it is much more likely that a non-regular network will result from an unbiased training process. In order to overcome this limitation, in the sequel we present the extension of our extraction algorithm to the general case, the case of non-regular networks. The idea is to investigate fragments of the non-regular network in order to find regularities over which the above described extraction algorithm could be applied. We would then split a non-regular network into regular subnetworks, extract the symbolic knowledge from each subnetwork, and finally assemble the rule set of the original non-regular network. That, however, is a decompositional approach, and we need to bear in mind that the collective behavior of a network is not equivalent to the behavior of its parts grouped together. We will need, therefore, to be specially careful when assembling the network's final set of rules.

The problem with non-regular networks is that it is difficult to find the ordering on the set of input vectors without having to actually check each input. In this case, the gain obtained in terms of complexity could be lost. By considering its regular subnetworks, the main problem we have to tackle is how to combine the information obtained into the network's rule set. That problem is due mainly to the non-discrete nature of the network's hidden neurons. As we have seen in Example 1, that is the reason why a decompositional approach may be unsound (see Section 2). In order to solve this problem, we will assume that hidden neurons present four possible activation values $(-1, A_{max}, A_{min}, 1)$. Performing a kind of worst case analysis, we will be able to show that the general case extraction is sound, although we will have to trade completeness for efficiency.

5.1. Regular subnetworks

We start by defining precisely the above intuitive concept of a subnetwork.

Definition 29 (*Subnetworks*). Let N be a neural network with p input neurons $\{i_1, \ldots, i_p\}$, r hidden neurons $\{n_1, \ldots, n_r\}$ and q output neurons $\{o_1, \ldots, o_q\}$. Let N' be a neural network with p' input neurons $\{i'_1, \ldots, i'_{p'}\}$, r' hidden neurons $\{n'_1, \ldots, n'_{r'}\}$ and q' output neurons $\{o'_1, \ldots, o'_{q'}\}$. N' is a *subnetwork* of N iff $0 \leqslant p' \leqslant p$, $0 \leqslant r' \leqslant r$, $0 \leqslant q' \leqslant q$, and for all i'_i, n'_j, o'_k in N', $W_{n'_j i'_i} = W_{n_j i_i}$, $W_{o'_k n'_j} = W_{o_k n_j}$, $\theta_{n'_j} = \theta_{n_j}$ and $\theta_{o'_k} = \theta_{o_k}$.

Our first task is to find the regular subnetworks of a non-regular network. It is not difficult to verify that any network containing a single hidden neuron is regular. As a result, we could be tempted to split a non-regular network with r hidden neurons into r subnetworks, each containing the same input and output neurons as the original network plus only one of its hidden neurons.

However, let us briefly analyze what could happen if we were to extract rules from each of the above subnetworks. Suppose that, for a given output neuron x, from the subnetwork containing hidden neuron n_1, the extraction algorithm obtains the rules $a, b \rightarrow_{n_1} x$ and $c, d \rightarrow_{n_1} x$, while from the subnetwork containing hidden neuron n_2, it obtains the rule $c, d \rightarrow_{n_2} x$. The problem is that the information that $[a, b] = (1, 1)$ activates x through n_1 is not very useful. It may be the case that the same input $[a, b] = (1, 1)$ has no effect on the activation of x through n_2, or that it actually blocks the activation of x through n_2. It may also be the case that, for example, $a, d \rightarrow x$ as a result of the combination of the activation values of n_1 and n_2, but not through each one of them individually. If, therefore, we take the intersection of the rules derived from each subnetwork, we would be extracting only the rules that are encoded in every hidden neuron individually, but not the rules derived from each hidden neuron or from the collective effect of the hidden neurons. If, on the other hand, we take the union of the rules derived from each subnetwork, then the extraction could clearly be unsound.

It seems that we need to analyze a non-regular network first from the input layer to each of the hidden neurons, and then from the hidden layer to each of the output neurons. That motivates the following definition of "*Basic Neural Structures*".

Definition 30 (*Basic Neural Structures*). Let N be a neural network with p input neurons $\{i_1, \ldots, i_p\}$, r hidden neurons $\{n_1, \ldots, n_r\}$ and q output neurons $\{o_1, \ldots, o_q\}$. A subnetwork N' of N is a *Basic Neural Structure* (*BNS*) iff either N' contains exactly p input neurons, 1 hidden neuron and 0 output neurons of N, or N' contains exactly 0 input neurons, r hidden neurons and 1 output neuron of N.

Note that a *BNS* is a neural network with no hidden neurons and a single neuron in its output layer. Note also that a network N with r hidden neurons and q output neurons contains $r + q$ *BNS*s. We call a *BNS* containing no output neurons of N, an *Input to Hidden BNS*; and a *BNS* containing no input neurons of N, a *Hidden-to-Output BNS*.

Proposition 31. *Any BNS is* (*vacuously*) *regular.*

Proof. Directly by Definition 30, by applying the Transformation Algorithm on a *BNS*, a network without complementary literals in the input layer is obtained. By Definition 14, since a *BNS* does not contain hidden neurons, it is (vacuously) regular. □

Proposition 31 shows that the Transformation Algorithm applied over a *BNS* will derive a positive network, the *BNS*'s positive form, which will not contain pairs of neurons labelled as complementary literals in its input layer. The above result indicates that *BNS*s, which can be easily obtained from a network N, are suitable subnetworks for applying the extraction algorithm when N is a non-regular network.

5.2. Knowledge extraction from BNSs

We have seen that, if we split a non-regular network into *BNS*s, there is always an ordering easily found in each subnetwork. The problem, now, is that *Hidden-to-Output*

*BNS*s do not present discrete activation values $\{-1, 1\}$ in their input layer. Instead, each input neuron may present activation in the ranges $(-1, A_{max})$ or $(A_{min}, 1)$, where $A_{max} \in (-1, 0)$ is a predefined value, and we will need to consider this during the extraction from *Hidden-to-Output BNS*s. For the time being, let us simply assume that each neuron in the input layer of a *Hidden-to-Output BNS* is labeled n_i, and if n_i is connected to the neuron in the output layer of the *BNS* through a negative weight, then we rename it $\sim n_i$ when applying the Transformation Algorithm, as done for regular networks. Moreover, let us assume that neurons in the input layer of the positive form of *Hidden-to-Output BNS*s present activation values -1 or A_{min} only. This results from the above mentioned worst case analysis, as we will see later in this section.

We need to rewrite Search Space Pruning Rules 1 and 2 for *BNS*s. Now, given a *BNS* with s input neurons $\{i_1, \ldots, i_s\}$ and the output neuron o_j, the constraint $\boldsymbol{C}_{o_j}$ on the activation of o_j for an input vector $\boldsymbol{i}$ is simply given by:

$$o_j \text{ is active for } \boldsymbol{i} \text{ iff } W_{o_j i_1} i_1 + W_{o_j i_2} i_2 + \cdots + W_{o_j i_s} i_s > h^{-1}(A_{min}) + \theta_{o_j}. \tag{5}$$

Proposition 32. *Let $\mathcal{I}_+$ be the set of input neurons of the positive form B_+ of a BNS B with output o_j. Let $\boldsymbol{I}_+ = \wp(\mathcal{I}_+)$ be ordered under the set inclusion relation $\leqslant_{\boldsymbol{I}_+}$, and $\boldsymbol{i}_m, \boldsymbol{i}_n \in \boldsymbol{I}_+$. If $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}_+} \boldsymbol{i}_n$ then $o_j(\sigma_{[\mathcal{I}_+]}(\boldsymbol{i}_m)) \leqslant o_j(\sigma_{[\mathcal{I}_+]}(\boldsymbol{i}_n))$ in B.*

Proof. If B is an *Input-to-Hidden BNS* then the proof is trivial, by Proposition 31 and Proposition 11. If B is a *Hidden-to-Output BNS*, assume $\boldsymbol{i}_m(i_k) = -1$ and $\boldsymbol{i}_n(i_k) = A_{min}$. Since all the weights in B_+ are positive real numbers and $A_{min} > 0$, we obtain $(W_{o_j i_k}(-1) - \theta_{o_j}) \leqslant (W_{o_j i_k}(A_{min}) - \theta_{o_j})$. Since $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}_+} \boldsymbol{i}_n$, we also have $(\sum_{i=1}^{p}(W_{o_j i_i}\boldsymbol{i}_m(i_i) - \theta_{o_j})) \leqslant (\sum_{i=1}^{p}(W_{o_j i_i}\boldsymbol{i}_n(i_i) - \theta_{o_j}))$, and by the monotonically crescent characteristic of $h(x)$, $h(\sum_{i=1}^{p}(W_{o_j i_i}\boldsymbol{i}_m(i_i) - \theta_{o_j})) \leqslant h(\sum_{i=1}^{p}(W_{o_j i_i}\boldsymbol{i}_n(i_i) - \theta_{o_j}))$, i.e., $o_j(\boldsymbol{i}_m) \leqslant o_j(\boldsymbol{i}_n)$ in B_+. Finally, from the definition of σ, mapping input vectors of B_+ into input vectors of B, it follows directly that $o_j(\sigma_{[\mathcal{I}_+]}(\boldsymbol{i}_m)) \leqslant o_j(\sigma_{[\mathcal{I}_+]}(\boldsymbol{i}_n))$ in B. □

Corollary 33 (BNS Pruning Rule 1). *Let $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}} \boldsymbol{i}_n$. If $\boldsymbol{i}_n$ does not satisfy the constraint $\boldsymbol{C}_{o_j}$ on a BNS's output neuron, then $\boldsymbol{i}_m$ does not satisfy $\boldsymbol{C}_{o_j}$ either.*

Proof. Directly from Proposition 32. □

Corollary 34 (BNS Pruning Rule 2). *Let $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}} \boldsymbol{i}_n$. If $\boldsymbol{i}_m$ satisfies the constraint $\boldsymbol{C}_{o_j}$ on a BNS's output neuron, then $\boldsymbol{i}_n$ also satisfies $\boldsymbol{C}_{o_j}$.*

Proof. Directly from Proposition 32. □

The particular characteristic of *BNS*s, specifically because they have no hidden neurons, allows us to define a new ordering that can be very useful in helping to reduce the search space of the extraction algorithm. If now, in addition, we take into account the values of the weights of the *BNS*, we may be able to assess, given two input vectors $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$

such that $\langle \boldsymbol{i}_m \rangle = \langle \boldsymbol{i}_n \rangle$, whether $o_j(\boldsymbol{i}_m) \leqslant o_j(\boldsymbol{i}_n)$ or vice-versa.[14] Assume, for instance, that $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ differ only on inputs i_i and i_k, where $i_i = 1$ in $\boldsymbol{i}_n$ and $i_k = 1$ in $\boldsymbol{i}_m$. Thus, if $|W_{o_j i_i}| \leqslant |W_{o_j i_k}|$ it is not difficult to see that $o_j(\boldsymbol{i}_n) \leqslant o_j(\boldsymbol{i}_m)$. Let us formalize this idea.

Proposition 35 (BNS Pruning Rule 3). *Let $\boldsymbol{i}_m, \boldsymbol{i}_n$ and $\boldsymbol{i}_o$ be three different input vectors in $\boldsymbol{I}$ such that dist$(\boldsymbol{i}_m, \boldsymbol{i}_o) = 1$, dist$(\boldsymbol{i}_n, \boldsymbol{i}_o) = 1$ and $\langle \boldsymbol{i}_m \rangle, \langle \boldsymbol{i}_n \rangle < \langle \boldsymbol{i}_o \rangle$, that is, both $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ are immediate predecessors of $\boldsymbol{i}_o$. Let $\boldsymbol{i}_m$ be obtained from $\boldsymbol{i}_o$ by flipping the ith input from* 1 (*respectively A_{min} for Hidden-to-Output BNSs*) *to* -1, *while $\boldsymbol{i}_n$ is obtained from $\boldsymbol{i}_o$ by flipping the kth input from* 1 (*respectively A_{min} for Hidden-to-Output BNSs*) *to* -1. *If $|W_{o_j i_k}| \leqslant |W_{o_j i_i}|$ then $o_j(\boldsymbol{i}_m) \leqslant o_j(\boldsymbol{i}_n)$. In this case, we write $\boldsymbol{i}_m \leqslant_{\langle\,\rangle} \boldsymbol{i}_n$.*

Proof. We know that both $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ are obtained from $\boldsymbol{i}_o$ by flipping, respectively, inputs $\boldsymbol{i}_o(i)$ and $\boldsymbol{i}_o(k)$ from 1 (respectively A_{min}) to -1. We also know that $o_j(\boldsymbol{i}_o) = h(W_{o_j i_i}\boldsymbol{i}_o(i) + W_{o_j i_k}\boldsymbol{i}_o(k) + \Delta + \theta_{o_j})$, and that $A_{min} > 0$. For *Input-to-Hidden BNS*s, $o_j(\boldsymbol{i}_m) = h(-W_{o_j i_i} + W_{o_j i_k} + \Delta + \theta_{o_j})$ and $o_j(\boldsymbol{i}_n) = h(W_{o_j i_i} - W_{o_j i_k} + \Delta + \theta_{o_j})$. For *Hidden-to-Output BNS*s, $o_j(\boldsymbol{i}_m) = h(-W_{o_j i_i} + A_{min}W_{o_j i_k} + \Delta + \theta_{o_j})$ and $o_j(\boldsymbol{i}_n) = h(A_{min}W_{o_j i_i} - W_{o_j i_k} + \Delta + \theta_{o_j})$. Since $|W_{o_j i_k}| \leqslant |W_{o_j i_i}|$, and from the monotonically crescent characteristic of $h(x)$, we obtain $o_j(\boldsymbol{i}_m) \leqslant o_j(\boldsymbol{i}_n)$ in both cases. □

As before, a direct result of Proposition 35 is that: if $\boldsymbol{i}_m$ satisfies the constraint $\boldsymbol{C}_{o_j}$ on the output neuron of the *BNS*, then $\boldsymbol{i}_n$ also satisfies $\boldsymbol{C}_{o_j}$. By contraposition, if $\boldsymbol{i}_n$ does not satisfy $\boldsymbol{C}_{o_j}$ then $\boldsymbol{i}_m$ does not satisfy $\boldsymbol{C}_{o_j}$ either.

Proposition 36 (BNS Pruning Rule 4). *Let $\boldsymbol{i}_m, \boldsymbol{i}_n$ and $\boldsymbol{i}_o$ be three different input vectors in $\boldsymbol{I}$ such that dist$(\boldsymbol{i}_m, \boldsymbol{i}_o) = 1$, dist$(\boldsymbol{i}_n, \boldsymbol{i}_o) = 1$ and $\langle \boldsymbol{i}_o \rangle < \langle \boldsymbol{i}_m \rangle, \langle \boldsymbol{i}_n \rangle$, that is, both $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ are immediate successors of $\boldsymbol{i}_o$. Let $\boldsymbol{i}_m$ be obtained from $\boldsymbol{i}_o$ by flipping the ith input from* -1 *to* 1 (*respectively A_{min} for Hidden-to-Output BNSs*), *while $\boldsymbol{i}_n$ is obtained from $\boldsymbol{i}_o$ by flipping the kth input from* -1 *to* 1 (*respectively A_{min} for Hidden-to-Output BNSs*). *If $|W_{o_j i_k}| \leqslant |W_{o_j i_i}|$, then $o_j(\boldsymbol{i}_n) \leqslant o_j(\boldsymbol{i}_m)$. In this case, we write $\boldsymbol{i}_n \leqslant_{\langle\,\rangle} \boldsymbol{i}_m$.*

Proof. This is the contrapositive of Proposition 35. □

Example 37. Consider the network N of Fig. 10(1) and its positive form N_+ at Fig. 10(2), obtained by applying the Transformation Algorithm over each *BNS* of N. N_+ contains three *BNS*s—two *Input-to-Hidden BNS*s, one with inputs $[a, b, c]$ and output n_1, and the other with inputs $[a, b, \sim c]$ and output n_2, and one *Hidden-to-Output BNS*, having inputs $[n_1, \sim n_2]$ and output x.

Considering the ordering on set inclusion, we verify that $[a, b, c] = (1, 1, 1)$ is the maximum element of the *BNS* with output n_1, $[a, b, \sim c] = (1, 1, 1)$ is the maximum element of the *BNS* with output n_2, and $[n_1 \sim n_2] = (A_{min}, A_{min})$ is the maximum element of the *BNS* with output x.

If now we add information about the weights, we can apply Pruning Rules 3 and 4 as well. Take, for example, the *BNS* with output n_1, where $|W_{n_1 b}| \leqslant |W_{n_1 c}| \leqslant$

[14] Recall that, previously, two input vectors $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ such that $\langle \boldsymbol{i}_m \rangle = \langle \boldsymbol{i}_n \rangle$ were incomparable.

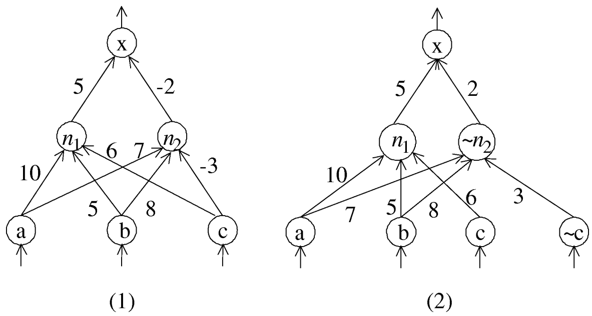


Fig. 10. A non-regular network (1) and its positive form (2) obtained by applying the Transformation Algorithm on its *BNS*s.

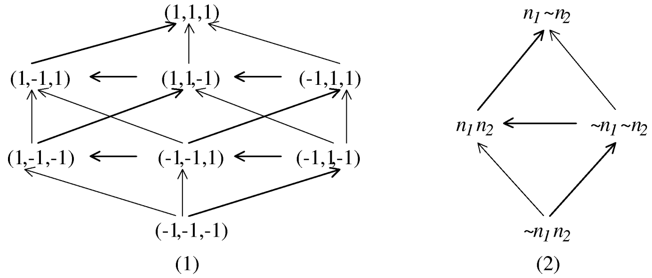


Fig. 11. Adding information about the weights of the *BNS*s with output n_1 (1) and x (2).

$|W_{n_1a}|$. Using Pruning Rules 3 and 4, we can obtain a new ordering on input vectors $\boldsymbol{i}_m$ and $\boldsymbol{i}_n$ such that $\langle \boldsymbol{i}_m\rangle = \langle \boldsymbol{i}_n\rangle$. We obtain $(-1, 1, 1) \leqslant_{\langle\,\rangle} (1, 1, -1) \leqslant_{\langle\,\rangle} (1, -1, 1)$ and $(-1, 1, -1) \leqslant_{\langle\,\rangle} (-1, -1, 1) \leqslant_{\langle\,\rangle} (1, -1, -1)$. Similarly, given $|W_{xn_2}| \leqslant |W_{xn_1}|$, we obtain $\{\sim n_1, \sim n_2\} \leqslant_{\langle\,\rangle} \{n_1, n_2\}$ for the *Hidden-to-Output BNS*.[15] Fig. 11 contains two diagrams in which this new ordering is superimposed on the previous set inclusion ordering for the *BNS*s with outputs n_1 and x.

The above example illustrates the ordering $\preceq$ on the set of input vectors $\boldsymbol{I}$ of *BNS*s. The ordering results from the superimposition of the ordering $\leqslant_{\langle\,\rangle}$, obtained from Pruning Rules 3 and 4, on the set inclusion ordering $\leqslant_{\boldsymbol{I}}$, obtained from Pruning Rules 1 and 2. Let us define $\preceq$ more precisely.

Definition 38. Let $\preceq$ be a partial ordering on the set of input vectors $\boldsymbol{I}$ of a *BNS*. For all $\boldsymbol{i}_m, \boldsymbol{i}_n \in \boldsymbol{I}$, $\boldsymbol{i}_m \preceq \boldsymbol{i}_n$ iff $\boldsymbol{i}_m \leqslant_{\boldsymbol{I}} \boldsymbol{i}_n$ or $\boldsymbol{i}_m \leqslant_{\langle\,\rangle} \boldsymbol{i}_n$.

Returning to Example 37, it is not difficult to see that the ordering $\preceq$ on the *BNS* with output n_1 is given by the diagram of Fig. 12 (see also Fig. 11(1)). Incomparable elements in $\preceq$, as $\boldsymbol{i}_1 = (1, -1, -1)$ and $\boldsymbol{i}_2 = (-1, 1, 1)$ at Fig. 12, indicate that it is not easy to establish

[15] Here, we have deliberately used $\{n_i, \sim n_i\}$, instead of $\{1, -1\}$, to stress the fact that hidden neurons do not present discrete activation values.

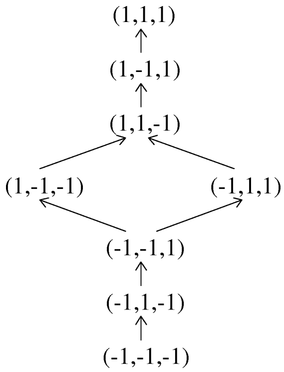


Fig. 12. The ordering $\preceq$ on the input vectors set of the *BNS* with output n_1.

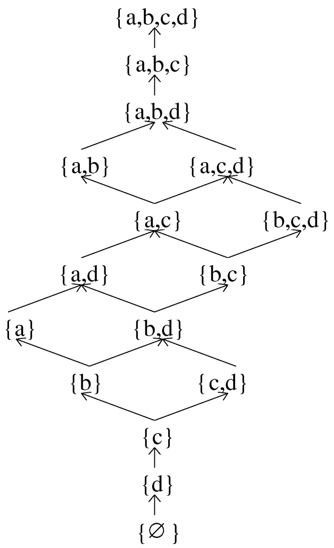


Fig. 13. $\preceq$ on $\wp(\mathcal{I})$, given $\mathcal{I} = \{a, b, c, d\}$ and $(1, 1, 1, 1) = [a, b, c, d]$.

whether $\boldsymbol{i}_1 \preceq \boldsymbol{i}_2$ without actually querying the *BNS* with both inputs. Note also that $\preceq$ is a chain for the *BNS* with output x, i.e., $\{\sim n_1, n_2\} \preceq \{\sim n_1, \sim n_2\} \preceq \{n_1, n_2\} \preceq \{n_1, \sim n_2\}$.

Fig. 13 displays $\preceq$ on $\boldsymbol{I} = \wp(\mathcal{I})$ for $\mathcal{I} = \{a, b, c, d\}$, given $(1, 1, 1, 1) = [a, b, c, d]$ and $|W_d| \leqslant |W_c| \leqslant |W_b| \leqslant |W_a|$. Note that $\preceq$ follows the ordering on $|W_a| + |W_b| + |W_c| + |W_d|$.

$\preceq$ provides a systematic way of searching the set of input vectors $\boldsymbol{I}$. Let us illustrate this with the following example, which also gives a glance about the implementation of the extraction algorithm in the general case.

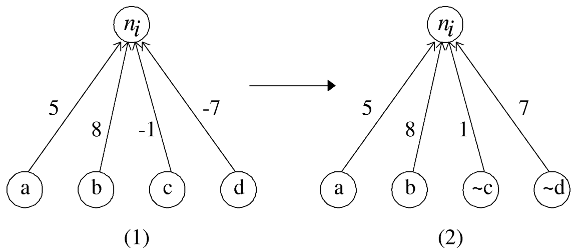


Fig. 14. An *Input-to-Hidden BNS* (1), and its positive form (2).

Example 39. Consider the *Input-to-Hidden BNS* of Fig. 14(1), and its positive form 14(2). The ordering's maximum element is input vector $\boldsymbol{i}_\top = (1,1,1,1) = (a, b, \sim c, \sim d)$. Taking the *BNS* of Fig. 14(2), if $\boldsymbol{i}_\top$ does not activate n_i then we proceed to generate the elements $\boldsymbol{i}_m$ such that $dist(\boldsymbol{i}_m, \boldsymbol{i}_\top) = 1$. However, Pruning Rule 3 says that there is an ordering among elements $\boldsymbol{i}_m$. For example, it says that $(1,1,1,-1) = (a, b, \sim c, d)$ provides a smaller activation value to n_i than $(1,1,-1,1) = (a, b, c, \sim d)$.

Therefore, given $W_{n_i \sim c} \leqslant W_{n_i a} \leqslant W_{n_i \sim d} \leqslant W_{n_i b}$, we start from $\boldsymbol{i}_\top$ by flipping from 1 to -1 the input $\sim c$ with the smallest weight $W_{n_i \sim c}$, and obtain the input vector $\boldsymbol{i}_1 = (1,1,-1,1)$. By Pruning Rule 3, the activation of n_i given $\boldsymbol{i}_1$ is greater than the activation of n_i given any other element $\boldsymbol{i}_m$ such that $\langle \boldsymbol{i}_m \rangle = \langle \boldsymbol{i}_1 \rangle$. Thus, if $n_i(\boldsymbol{i}_1) \leqslant A_{min}$ then $n_i(\boldsymbol{i}_m) \leqslant A_{min}$. In this case we could stop the search. Otherwise, we would have to derive the rule $a, b, c, \sim d \to n_i$, and carry on generating and querying the remaining elements $\boldsymbol{i}_m$ such that $\langle \boldsymbol{i}_m \rangle = \langle \boldsymbol{i}_1 \rangle$. Again, due to Pruning Rule 3, we could do so by flipping, from $\boldsymbol{i}_\top$, the input a with the next smallest weight, $W_{n_i a}$, and repeat the above process until either we can stop or we flip the input b with the largest weight, $W_{n_i b}$.

Similarly, starting from the ordering's minimum element $\boldsymbol{i}_\perp = (-1,-1,-1,-1) = (\sim a, \sim b, c, d)$, if $\boldsymbol{i}_\perp$ does not activate n_i then we flip from -1 to 1 the input $\sim c$ with the smallest weight $W_{n_i \sim c}$, to obtain input vector $\boldsymbol{i}_2 = (-1,-1,1,-1)$. By Pruning Rule 4, if $n_i(\boldsymbol{i}_2) > A_{min}$ then $n_i(\boldsymbol{i}_n) > A_{min}$ for all $\boldsymbol{i}_n$ such that $\langle \boldsymbol{i}_n \rangle = \langle \boldsymbol{i}_2 \rangle$. In this case, we could derive the rule $1(a, b, \sim c, \sim d) \to n_i$, using simplification *M of N*, and stop the search. Otherwise, we would need to generate another element from $\boldsymbol{i}_\perp$, this time by flipping the input a with the next smallest weight, and repeat the above process until either we can stop or we flip the input b with the largest weight, $W_{n_i b}$.

A systematic way of searching the set of input vectors $\boldsymbol{I}$ is obtained as follows. Given the maximum element, we order it from left to right with respect to the weights associated with each input, such that inputs with greater weights are on the left of inputs with smaller weights. In Example 39, we rearrange $(a, b, \sim c, \sim d)$ and obtain $(1,1,1,1) = [b, \sim d, a, \sim c]$. The search proceeds by flipping the right most input, then the second right most input and so on. At distance 2 from $\sup(\boldsymbol{I})$ and beyond, we only flip the inputs on the left of the left most input -1. In this way, we avoid repeating input vectors. Fig. 15 illustrates this process for the *BNS* of Example 39.

Similarly, starting from the minimum element, we rearrange $(\sim a, \sim b, c, d)$ and obtain $(-1,-1,-1,-1) = [\sim b, d, \sim a, c]$. Fig. 16 illustrates the process for the *BNS* of

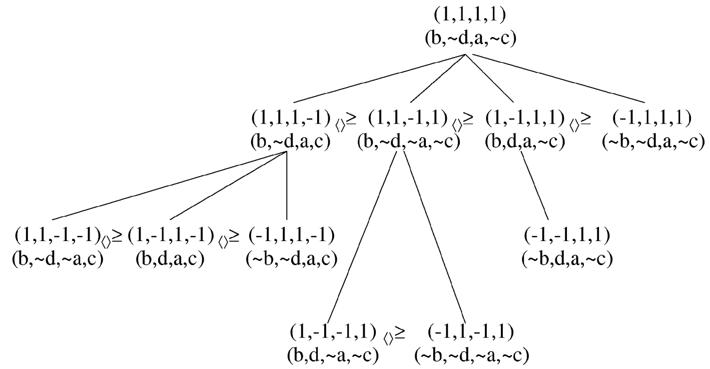


Fig. 15. Systematically deriving input vectors from $\boldsymbol{i}_\top$ without repetitions.

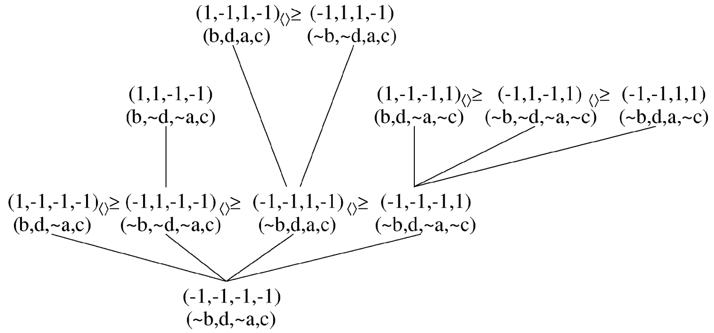


Fig. 16. Systematically deriving input vectors from $\boldsymbol{i}_\perp$ without repetitions.

Example 39. Now, at distance 2 from $\inf(\boldsymbol{I})$ and beyond, we only flip the inputs on the left of the left most input 1.

Note the symmetry between Figs. 15 and 16, reflecting, respectively, the use of Pruning Rules 3 and 4. Starting from $\sup(\boldsymbol{I})$, flipping the input with the smallest weight results in the next greatest input, while from $\inf(\boldsymbol{I})$, flipping the input with the smallest weight results in the next smallest input. Note also that the sequence in which the input vectors are generated, according to Figs. 15 and 16, complies with the ordering $\preceq$ on the set of input vectors, shown in Fig. 13.

Let us now focus on the problem of knowledge extraction from *Hidden-to-Output BNS*s. The problem lies on the fact that hidden neurons do not present discrete activation. As a result, we need to provide a special treatment for the procedure of knowledge extraction from *Hidden-to-Output BNS*s. We have seen already that, if we simply assume that hidden neurons are either fully active or non-active, then the extraction algorithm looses soundness.

We say that a hidden neuron is *active* if its activation value lies in the interval $(A_{min}, 1)$, or *non-active* if its activation value lies in the interval $(-1, A_{max})$.[16] Trying to find an

[16] Recall that $A_{min} \in (0, 1)$ and $A_{max} \in (-1, 0)$.

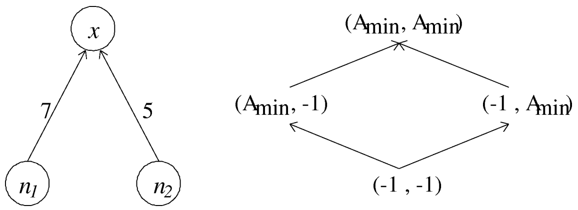


Fig. 17. A *Hidden-to-Output BNS* and the corresponding set inclusion ordering on the activation values of the hidden neurons in the worst cases.

ordering on such intervals of activation is not easy. For example, taking the *Hidden-to-Output BNS* of the network of Fig. 10(1), one can not say that having $n_1 < A_{max}$ and $n_2 < A_{max}$ results in a smaller activation value for x than having $n_1 < A_{max}$ and $n_2 > A_{min}$. This is so because, if $A_{max} = -A_{min} = -0.2$ then $n_1 = -0.3$ and $n_2 = -0.3$ may provide a greater activation in x than $n_1 = -0.95$ and $n_2 = 0.25$.

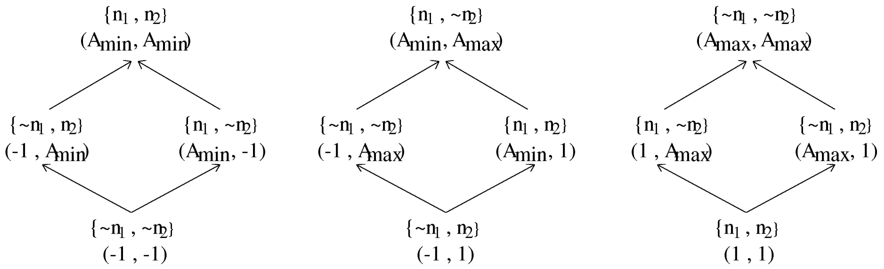
At this stage, we need to compromise in order to keep soundness. Roughly, we have to analyze the activation values of the hidden neurons in the "worst cases". Those activation values are given by -1 and A_{min} in the case of a hidden neuron connected through a positive weight to the output, and by A_{max} and 1 in the case of a hidden neuron connected through a negative weight to the output.

Example 40. Consider the *Hidden-to-Output BNS* of Fig. 17. The intuition behind its corresponding ordering is as follows: either both n_1 and n_2 present activation greater than A_{min}, or one of them presents activation greater than A_{min} while the other presents activation smaller than A_{max}, or both of them present activation smaller than A_{max}.

Considering the activation values in the worst cases, since the weights from n_1 and n_2 to x are both positive, if the activation of n_i is smaller than A_{max} then we assume that it is -1. On the other hand, if the activation of n_i is greater than A_{min}, then we consider that it is equal to A_{min}. In this way, we can derive the ordering of Fig. 17 safely, as we show in the sequel. In addition, given that $W_{xn_2} \leqslant W_{xn_1}$, we obtain $(-1, A_{min}) \preceq (A_{min}, -1)$. As before, in this case $\preceq$ is a chain.

The recipe for performing a sound extraction from non-regular networks, concerning *Hidden-to-Output BNS*s, is: *If the weight from n_i to o_j is positive then assume $n_i = A_{min}$ and $\sim n_i = -1$. If the weight from n_i to o_j is negative then assume $n_i = 1$ and $\sim n_i = A_{max}$.* These are the worst case analyses, which means that we consider the minimum contribution of each hidden neuron to the activation of an output neuron.

Remark 2. Note that when we consider that the activation values of hidden neurons are either positive in the interval $(A_{min}, 1)$ or negative in the interval $(-1, A_{max})$, we assume, without loss of generality, that the network's learning algorithm is such that no hidden neuron presents activation in the range $[A_{max}, A_{min}]$ (see [5]). Alternatively, one may assume that $A_{max} \simeq 0$ and $A_{min} \simeq 0$.



Case 1 *Case 2* *Case 3*

Fig. 18. Orderings on *Hidden-to-Output BNS*s with two input neurons n_1 and n_2, using worst case analyses on $(-1, A_{max})$ and $(A_{min}, 1)$.

In the sequel, we exemplify how to obtain the ordering on a *Hidden-to-Output BNS* with two input neurons n_1 and n_2, connected to an output neuron x with positive and negative weights.

We start by applying the Transformation Algorithm. We obtain the *BNS*'s positive form and check the labels of its input neurons (the network's hidden neurons). If they are labeled n_1 and n_2 ($\sup(\boldsymbol{I}) = (n_1, n_2)$) then the weights from both of them to x are positive. Thus, we assume that $\sim n_i = -1$ and $n_i = A_{min}$ for $i = \{1, 2\}$. As a result, we derive the ordering of Fig. 18(Case 1). If, however, the Transformation Algorithm tells us that $\sup(\boldsymbol{I}) = (n_1, \sim n_2)$ then we consider $\sim n_1 = -1$ and $n_1 = A_{min}$ for the activation values of n_1, and $\sim n_2 = A_{max}$ and $n_2 = 1$ for the activation values of n_2. Fig. 18(Case 2) shows the ordering obtained if $\sup(\boldsymbol{I}) = (n_1, \sim n_2)$. Finally, if $\sup(\boldsymbol{I}) = (\sim n_1, \sim n_2)$, we assume that $\sim n_i = A_{max}$ and $n_i = 1$ for $i = \{1, 2\}$, as shown in Fig. 18(Case 3). If, in addition, we have $|W_{o_j n_2}| \leqslant |W_{o_j n_1}|$, we also obtain $(A_{min}, -1) \leqslant_{\langle\,\rangle} (-1, A_{min})$ in Fig. 18(Case 1), $(A_{min}, 1) \leqslant_{\langle\,\rangle} (-1, A_{max})$ in Fig. 18(Case 2), and $(A_{max}, 1) \leqslant_{\langle\,\rangle} (1, A_{max})$ in Fig. 18(Case 3). Thus, the resulting orders $\preceq$ are chains, as expected. Note that the orders of Fig. 18 are valid for the original *BNS*s, and not for their positive forms.

Let us now see if we can define a mapping for *Hidden-to-Output BNS*s, analogous to the mapping σ for Regular Networks and *Input-to-Hidden BNS*s. In fact, if we assume, without loss of generality, that $A_{max} = -A_{min}$ then the same function σ mapping input vectors of the positive form into input vectors of the *BNS* can be used here. Let $i_i \in \{-1, A_{min}\}$, $i'_i \in \{-1, -A_{min}, A_{min}, 1\}$, $x_i \in \mathcal{I}_+, 1 \leqslant i \leqslant p$. Recall that $\sigma_{[x_1,\ldots,x_p]}(i_1, \ldots, i_p) = (i'_1, \ldots, i'_p)$, where $i'_i = i_i$ if x_i is a positive literal, and $i'_i = -i_i$ otherwise. Thus, $\sigma_{[a,\,\sim b, c,\,\sim d]}(A_{min}, A_{min}, -1, -1) = (A_{min}, -A_{min}, -1, 1)$. The following example illustrates the use of σ for *Hidden-to-Output BNS*s.

Example 41. Consider a *Hidden-to-Output BNS* (B) with three input neurons (n_1, n_2, n_3) and output o. Let $W_{on_1} > 0$, $W_{on_2} < 0$ and $W_{on_3} > 0$. Thus, the positive form (B^+) of B contains n_1, $\sim n_2$ and n_3 as input neurons. Using the mapping σ above, we obtain $\sigma_{[n_1,\,\sim n_2, n_3]}(A_{min}, A_{min}, A_{min}) = (A_{min}, -A_{min}, A_{min})$. In other words, querying the original *BNS* (B) with $[n_1, n_2, n_3] = (A_{min}, -A_{min}, A_{min})$ is equivalent to querying its positive form (B^+) with $[n_1, \sim n_2, n_3] = (A_{min}, A_{min}, A_{min})$. Similarly,

$\sigma_{[n_1,\sim n_2,n_3]}(-1,-1,-1) = (-1,1,-1)$, $\sigma_{[n_1,\sim n_2,n_3]}(-1,-1,A_{min}) = (-1,1,A_{min})$, and so on. As a result, since we have taken the activation values in the worst cases, the extraction process can be carried out by querying the positive form of the *BNS* with values in $\{-1, A_{min}\}$ only. In this way, the only difference between B^+ and the positive form of an *Input-to-Hidden BNS* is that input values 1 should be replaced by A_{min}. For example, in Figs. 15 and 16, it is sufficient to replace any input 1 by A_{min}, when considering a *Hidden-to-Output BNS.*

We are finally in a position to present the extraction algorithm extended for non-regular networks.

Knowledge Extraction Algorithm—General case

(1) Split the neural network N into *BNS*s;
(2) For each *BNS* $\mathcal{B}_i$ $(1 \leqslant i \leqslant r+q)$ do:
   (a) Apply the *Transformation Algorithm* and find its positive form $\mathcal{B}_i^+$;
   (b) Order $\mathcal{I}_+$ according to the weights associated with each input of $\mathcal{B}_i^+$;
   (c) If $\mathcal{B}_i^+$ is an *Input-to-Hidden BNS*, take $i_i \in \{-1, 1\}$;
   (d) If $\mathcal{B}_i^+$ is a *Hidden-to-Output BNS*, take $i_i \in \{-1, A_{min}\}$;
   (e) Find $\mathrm{Inf}(\boldsymbol{I})$ and $\mathrm{Sup}(\boldsymbol{I})$ with respect to $\mathcal{B}_i^+$, using σ;
   (f) Call the *Knowledge Extraction Algorithm for Regular Networks*, step (3), where $N_+ := \mathcal{B}_i^+$;
   /∗ *Recall that, now, we have to replace Search Space Pruning Rules* 1 *and* 2, *respectively, by BNS Pruning Rules* 1 *and* 2.
   /∗ *We also need to add the following lines to the extraction algorithm for regular networks* (*step* (3d)):
   - If BNS Pruning Rule 4 is applicable, stop generating the successors of $\boldsymbol{i}_\perp$;
   - If BNS Pruning Rule 3 is applicable, stop generating the predecessors of $\boldsymbol{i}_\top$;
(3) Assemble the final Rule Set of N.

In what follows, we describe in detail step (3) of the above algorithm, and discuss the problems resulting from the worst case analysis of *Hidden-to-Output BNS*s.

5.3. Assembling the final rule set

Steps (1) and (2) of the general case extraction algorithm generate local information about each hidden and output neuron. In step (3), such information needs to be carefully combined, in order to derive the final set of rules of N. We use n_i and $\sim n_i$ to indicate, respectively, that the activation of hidden neuron n_i is greater than A_{min} or smaller than A_{max}. Bear in mind, however, that hidden neurons n_i do not have concepts directly associated to them. Thus, the task of assembling the final set of rules is that of relating the concepts in the network's input layer directly to the ones in its output layer, by removing literals n_i. The following Lemma 42 will serve as basis for this task.

Lemma 42. *The extraction of rules from Input-to-Hidden BNSs is sound and complete.*

Proof. From Proposition 31 and Theorem 27, we obtain soundness of the rule set. From Proposition 31 and Theorem 28, we obtain completeness of the rule set. □

Lemma 42 allows us to use the *completion* of the rules extracted from *Input-to-Hidden BNS*s to assemble the set of rules of the network, i.e., it allows an extracted rule of the form $X_1, \ldots, X_p \to L_j$ to be substituted by the stronger $X_1, \ldots, X_p \leftrightarrow L_j$. For example, assume that the extraction algorithm derives $a \to n_1$ from a *BNS* $\mathcal{B}_1$ and $b{\sim}c \to n_2$ from a *BNS* $\mathcal{B}_2$. By Lemma 42, we have $a \leftrightarrow n_1$ and $b{\sim}c \leftrightarrow n_2$. By contraposition, we have ${\sim}a \leftrightarrow {\sim}n_1$ from $\mathcal{B}_1$, and ${\sim}b \vee c \leftrightarrow {\sim}n_2$ from $\mathcal{B}_2$. Now that we have the necessary information regarding the activation values of n_1 and n_2, assume that we have derived the rule $n_1{\sim}n_2 \to x$ from a *Hidden-to-Output BNS* $\mathcal{B}_3$. We know that $a \to n_1$ and ${\sim}b \vee c \to {\sim}n_2$. As a result, we may assemble the final set of rules regarding output x: $\{a{\sim}b \to x,\ ac \to x\}$.

The following example illustrates how to assemble the final set of rules of a network in a sound mode. It also illustrates the incompleteness of the general case extraction, which we prove in the sequel.

Example 43. Consider a neural network N with two input neurons a and b, two hidden neurons n_1 and n_2 and one output neuron x. Assume that the set of weights is such that the activation values in the table below are obtained for each input vector.

a	b	n_1	n_2	x
-1	-1	$< A_{max}$	$< A_{max}$	$< A_{max}$
-1	1	$> A_{min}$	$> A_{min}$	$< A_{max}$
1	-1	$< A_{max}$	$< A_{max}$	$< A_{max}$
1	1	$> A_{min}$	$< A_{max}$	$> A_{min}$

An exhaustive pedagogical extraction algorithm, although inefficiently, would derive the unique rule $ab \to x$ from N. That is because $[a, b] = (1, 1)$ is the only input vector that activates x. A decompositional approach, on the other hand, would split the network into its *BNS*s. Since $[a, b] = (-1, 1)$ and $[a, b] = (1, 1)$ activate n_1, the rules ${\sim}ab \to n_1$ and $ab \to n_1$ would be derived, and hence $b \to n_1$. Similarly, the rule ${\sim}ab \to n_2$ would be derived, since $[a, b] = (-1, 1)$ also activates n_2.

Taking $A_{min} = 0.5$, suppose that, given $[a, b] = (-1, 1)$, the activation values of n_1 and n_2 are, respectively, 0.6 and 0.95. As we have seen in Example 1, if we had assumed that the activation values of n_1 and n_2 were both 1, we could have wrongly derived the rule $n_1 n_2 \to x$ (unsoundness). To solve this problem, we have taken the activation values of the hidden neurons in the worst case, namely, $n_1 = A_{min}$ and $n_2 = A_{min}$.

Now, given $[a, b] = (1, 1)$, suppose that the activation values of n_1 and n_2 are, respectively, 0.9 and -0.6. If we take the activation values in the worst case, that is, $n_1 = A_{min}$ and $n_2 = -1$, we might not be able to derive the rule $n_1{\sim}n_2 \to x$, as expected (incompleteness).

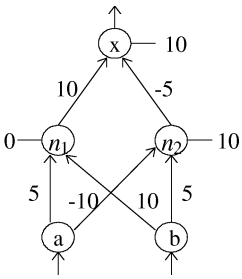


Fig. 19.

Finally, once we have managed to derive the rule $n_1 {\sim} n_2 \to x$ from the *Hidden-to-Output BNS* of N, possibly by fine-tuning the value of A_{min} in the extraction algorithm, the final set of rules of N can be assembled as follows: by Lemma 42, we derive $b \leftrightarrow n_1$ and ${\sim}a \wedge b \leftrightarrow n_2$. From ${\sim}a \wedge b \leftrightarrow n_2$, we obtain $a \vee {\sim}b \leftrightarrow {\sim}n_2$. From $b \leftrightarrow n_1$, $a \vee {\sim}b \leftrightarrow {\sim}n_2$ and $n_1 {\sim} n_2 \to x$, we have $b \wedge (a \vee {\sim}b) \to x$, which is equivalent to $ab \to x$, in accordance with the result of the exhaustive pedagogical extraction. A neural network that presents the activation values used in this example is given in Fig. 19.

Lemma 44. *The extraction of rules from Hidden-to-Output BNSs is sound.*

Proof. From Proposition 31 and Theorem 27, if we are able to derive a rule r taking $n_i \in \{-1, A_{min}\}$ then, from the monotonically crescent characteristic of $h(x)$, r will still be valid if $n_i \in \{(-1, -A_{min}), (A_{min}, 1)\}$, where $A_{min} > 0$. □

Theorem 45. *The extraction algorithm for non-regular networks is sound.*

Proof. Directly from Lemmas 42 and 44. □

Theorem 46. *The extraction algorithm for non-regular networks is incomplete.*

Proof. We give a counter-example. Let $\mathcal{B}$ be a *Hidden-to-Output BNS* with input n_1 and output x. Let $\beta = 1$, $W_{xn_1} = 1$, $\theta_x = 0.1$. Assume $A_{min} = 0.4$. Given $Act(n_1) = 1$, we obtain $Act(x) = 0.42$, i.e., $n_1 \to x$. Taking $Act(n_1) = A_{min}$, we obtain $Act(x) = 0.15$ and, thus, we have lost $n_1 \to x$. □

As far as efficiency is concerned, one can apply the extraction algorithm until a predefined number of input vectors is queried, and then test the accuracy of the set of rules derived against the accuracy of the network. If, for instance, in a particular application, the set of rules obtained classifies correctly, say, 95% of the training and testing examples of the network, then one could stop the extraction process. Theorem 45 will ensure that it is sound.

6. Experimental results

In this section, we successfully apply the above method of rule extraction from trained networks in well known traditional examples and real-world application problems. The implementation of the system has been kept as simple as possible, and does not benefit from all the features of the theory presented above.

Our purpose in this section is to show that the implementation of a sound method of extraction can be efficient, and to confirm the importance of extracting nonmonotonic theories from trained networks. Our intention is not to provide an exhaustive comparative analysis with other extraction methods. Such a comparison could be easily biased, depending on the application at hand, training parameters and testing methodology used. Nevertheless, in what follows, we also present the results reported in [10,30,34], when available.

We have used three application domains in order to test the extraction algorithm: the MONK's problems [32], DNA sequences analysis [5,10,30,34], and Power Systems FAULT DIAGNOSIS [4,31]. Briefly, the results obtained indicate that a very high fidelity between the network and the extracted set of rules can be achieved. They also indicate that a reduced readability is the price one has to pay for soundness. We will discuss this problem in detail in Section 6.4.

The extraction system consists of three modules: its main module takes a trained neural network (its set of weights and activation functions), searches the set of input vectors and generates a set of rules accordingly, another module simplifies the set of rules, and yet another checks its accuracy against that of the network, given a test set, and the fidelity of the set of rules to the network. The system was implemented in ANSI C (5K lines of code) and is available upon request. Implementation details will be discussed in another paper. We start by presenting two very simple examples, which will help the reader to recall the sequence of operations contained in the extraction process.

Example 47 (*The XOR problem*). A network with p input neurons, q hidden neurons and r output neurons contains q *Input-to-Hidden BNS*s, each with p inputs and a single output, and r *Hidden-to-Output BNS*s, each with q inputs and a single output. To each *BNS* we apply a transformation whereby we rename input neurons x_k linked through negative weights to the output, by $\sim x_k$ and replace each weight $W_{lk} \in \mathbb{R}$ by its modulus. We call the result the *positive form* of the *BNS*. For example, in Fig. 20, N_1 and N_2 are the positive forms of the *Input-to-Hidden BNS*s of N, while N_3 is the positive form of the *Hidden-to-Output BNS* of N. We then define the function σ mapping input vectors of the positive form into input vectors of the *BNS*. For example, for N_1 $\sigma_{[a,\sim b]}(1,1) = (1,-1)$.

Given a 2-*ary* input vector, $\preceq$ is a linear ordering. For N_1, $(-1,-1) \preceq (1,-1) \preceq (-1,1) \preceq (1,1)$, and for N_2, $(-1,-1) \preceq (-1,1) \preceq (1,-1) \preceq (1,1)$, where $(1,1) = [a, \sim b]$ in both. Querying N_1, h_0 is active for $(1,1)$ only. Thus, by applying σ we derive $a{\sim}b \to h_0$. Querying N_2, h_1 is not active for $(-1,-1)$ only. Similarly, we derive $ab \to h_1$, ${\sim}a{\sim}b \to h_1$ and $a{\sim}b \to h_1$. The last two rules can be simplified to obtain ${\sim}b \to h_1$, since ${\sim}b$ implies h_1 given either a or ${\sim}a$. Similarly, from $ab \to h_1$ and $a{\sim}b \to h_1$ we obtain $a \to h_1$.

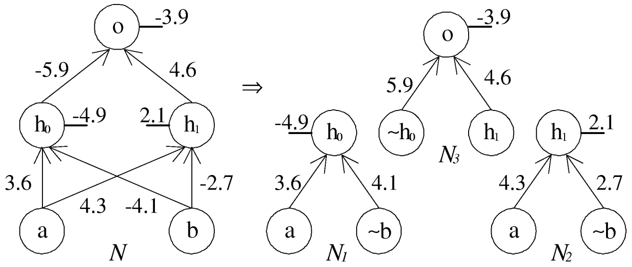


Fig. 20. The network N, having tanh as activation function, computes $\overline{XOR}$. We will extract rules for h_0, h_1 and o by querying N_1, N_2 and N_3, respectively, and then assemble the set of rules of N.

Considering now *Hidden-to-Output BNS*s, it is usually assumed that the network's hidden neurons present discrete activation values such as $\{-1, 1\}$. We know however that this is not the case, and therefore problems may arise from such assumption. At this point we need to compromise. Either we assume that the activation values of the hidden neurons are in $\{-1, A_{min}\}$, and then are able to show that the extraction is sound, but incomplete, or we assume that they are in $\{-A_{min}, 1\}$, obtaining an unsound, but complete, extraction. We have chosen the first approach.[17] For N_3 we have $(-1, -1) \preceq (-1, A_{min}) \preceq (A_{min}, -1) \preceq (A_{min}, A_{min})$, where $(A_{min}, A_{min}) = [\sim h_0, h_1]$ and $A_{min} = 0.5$. Only (A_{min}, A_{min}) activates o, and we derive the rule $\sim h_0 h_1 \rightarrow o$.

Finally, to assemble the rule set of N, we take the *completion* of each rule extracted from *Input-to-Hidden BNS*s. We have $a{\sim}b \rightarrow h_0$, $a \rightarrow h_1$, $\sim b \rightarrow h_1$ and $\sim h_0 h_1 \rightarrow o$. And from $a{\sim}b \leftrightarrow h_0$ and $a \vee \sim b \leftrightarrow h_1$ we obtain $(\sim a \vee b) \wedge (a \vee \sim b) \rightarrow o$; the $\overline{XOR}$ function.

Example 48 (EXACTLY 1 OUT OF 5). We train a network with five input neurons $\{a, b, c, d, e\}$, two hidden neurons $\{h_0, h_1\}$ and one output neuron $\{o\}$, on all the 32 possible input vectors. The network's output neuron fires iff exactly one of its inputs fires. Although this is a very simple network, it is not straightforward to verify, by inspecting its weights, that it computes the following rule: "*Exactly* 1 *out of* $\{a, b, c, d, e\}$ *implies* o".

Assume the following order on the weights linking the input layer to each hidden neuron h_o and h_1: $|W_{h_0d}| \leqslant |W_{h_0e}| \leqslant |W_{h_0c}| \leqslant |W_{h_0a}| \leqslant |W_{h_0b}|$ and $|W_{h_1d}| \leqslant |W_{h_1e}| \leqslant |W_{h_1a}| \leqslant |W_{h_1c}| \leqslant |W_{h_1b}|$. We split the network into its *BNS*s and apply the extraction algorithm. Taking $\mathcal{I} = [a, b, c, d, e]$ for the *BNS* with output h_0, we find out that input $(-1, -1, -1, 1, -1)$ activates h_0, by querying the *BNS*. Since $|W_{h_0d}|$ is the smallest weight, from the ordering $\preceq$ on $\boldsymbol{I}$ and by applying Definitions 20 and 22, we derive the rule $1(abcde) \rightarrow h_0$. Note that, by Definition 23, this rule subsumes $m(abcde) \rightarrow h_0$, for $m > 1$. Taking again $\mathcal{I} = [a, b, c, d, e]$ but now for the *BNS* with output h_1, we find out that input $(-1, -1, -1, 1, 1)$ activates h_1. Similarly, from the ordering $\preceq$ on $\boldsymbol{I}$ and by

[17] Here, we perform the worst case analysis. By choosing activations in $\{-1, A_{min}\}$, misclassifications occur because of the absence of a rule (incompleteness). Analogously, by choosing $\{-A_{min}, 1\}$, misclassifications are due to the inappropriate presence of rules in the rule set (unsoundness). In this context, the choice of $\{-1, 1\}$ yields unsound and incomplete rule sets.

applying Definitions 20 and 22, we derive the rule $2(abcde) \to h_1$. Finally, for the *Hidden-to-Output BNS*, $\mathcal{I} = [h_0, \sim h_1]$. Taking $A_{min} = 0.5$, o is only activated by (A_{min}, A_{min}) and we derive the rule $h_0 \sim h_1 \to o$.

In order to obtain the rule mapping inputs $\{a, b, c, d, e\}$ directly into the output $\{o\}$, we take the *completion* of the rules extracted from *Input-to-Hidden BNS*s: $1(abcde) \leftrightarrow h_1$ and $2(abcde) \leftrightarrow h_2$. Therefore, "*Exactly* 1 *out of* $\{a, b, c, d, e\}$ *implies* o" is obtained by computing $1(abcde) \wedge \sim 2(abcde) \to o$, i.e., "*At least* 1 *out of* $\{a, b, c, d, e\}$ AND *At most* 1 *out of* $\{a, b, c, d, e\}$ *implies* o". Note that a network with a single hidden neuron would not be able to learn such a rule.

In what follows, we briefly describe each of the above mentioned applications, and present the results of the extraction algorithm. For each problem, we investigate three parameters: the *accuracy* of the set of rules against that of the network with respect to a test set, the *fidelity* of the set of rules to the network, i.e., its ability to mimic the network's behavior, and the *readability* of the set of rules in terms of its size.

6.1. The MONK's problems

As a point of departure for testing, we applied the extraction algorithm to the Monk's problems [32]: three examples which have been used as benchmark for performance comparison between a range of symbolic and connectionist machine learning systems. Briefly, in the Monk's problems, robots in an artificial domain are described by six attributes with the following possible values:

(1) *head_shape* {round, square, octagon},
(2) *body_shape* {round, square, octagon},
(3) *is_smiling* {yes, no},
(4) *holding* {sword, balloon, flag},
(5) *jacket_color* {red, yellow, green, blue}, and
(6) *has_tie* {yes, no}.

Problem 1 trains a network with 124 examples, selected from 432, where (*head_shape* = *body_shape*) $\vee$ (*jacket_color* = *red*). Problem 2 trains a network with 169 examples, selected from 432, where *exactly two of the six attributes have their first value*. Problem 3 trains a network with 122 examples with 5% noise, selected from 432, where (*jacket_color* = *green* $\wedge$ *holding* = *sword*) $\vee$ (*jacket_color* $\neq$ *blue* $\wedge$ *body_shape* $\neq$ *octagon*). The remaining examples are used in the respective test sets.

We use the same architectures as Thrun [32], i.e., single hidden layer networks with three, two and four hidden neurons, for Problems 1, 2 and 3, respectively; 17 input neurons, one for each attribute value, and a single output neuron, for the binary classification task. We use the standard backpropagation learning algorithm [27]. All networks have been trained for 5,000 epochs, with an epoch being defined as one pass through the whole training set. Differently from Thrun, we use bipolar activation function, inputs in the set $\{-1, 1\}$, and $A_{min} = 0$ (see [5] for the motivation behind this).

For Problems 1, 2 and 3, the performance of the networks with respect to their test sets was 100%, 100% and 93.2%, respectively. The accuracy of the extracted sets of rules, in the same test sets, was 100%, 99.2% and 93.5%. The fidelity of the sets of rules to the

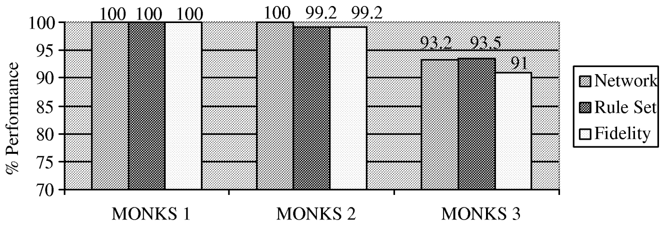


Fig. 21. The accuracy of the network, the accuracy of the extracted rule set and the fidelity of the rule set to the network with respect to the test sets of the Monk's Problems 1, 2 and 3, respectively.

networks was 100%, 99.2% and 91%. Fig. 21 displays the accuracy of the network, the accuracy of the set of rules, and the fidelity of the set of rules to the network, grouped for each problem.

The accuracy of the sets of rules is very similar to that of the networks. In Problem 1, the rule set matches exactly the behavior of the network. In Problem 2, the rule set fails to classify correctly two examples, and in Problem 3 the rule set classifies correctly one example wrongly classified by the network. Such differences are due to the incompleteness of the extraction algorithm.

Tables 1, 2, and 3 present, for Problems 1, 2, and 3, the number of input vectors queried during extraction and the number of rules obtained before and after simplifications *Complementary Literals* and *Subsumption* are applied. For example, for hidden neuron h_0 in Monk's Problem 1, 18,724 input vectors are queried generating 9455 rules that after simplification are reduced to 2633 rules. In general, less than 30% of the set of input vectors is queried and, among these, less than 50% generate rules.

In general, *Complementary Literal* and *Subsumption* reduce the rule set by 80%. *M of N* and *M of N Subsumption* further enhance the rule set readability. In particular, the rule set for Problem 1 is presented in Table 4. For short, we name each attribute value with a letter from a to q in the sequence presented above, such that $a = (\mathit{head\_shape} = \mathit{round})$, $b = (\mathit{head\_shape} = \mathit{square})$, and so on. We also use the *Integrity Constraints* of the Monk's Problems in order to present a clearer set of rules. For example, we do not present derived rules where *has_tie* = *yes* and *has_tie* = *no* simultaneously, although the network has generalized to include some of these rules.

By looking at the set of rules extracted and the much simpler description of Monk's Problem 1, it is clear that neural networks do not learn rules in a simple and structured way. Instead, they use a complex and redundant way of encoding rules. Not surprisingly, such a redundant representation is responsible for the network's robustness.

It is interesting that because the rule obtained for the *Hidden-to-Output BNS* of Monk's Problem 1 was $\sim h_1 \sim h_2 \rightarrow o$, and since the set of rules presents 100% of accuracy, hidden neuron h_0 is not necessary at all, i.e., the problem could have been solved by a network with two hidden neurons only, obtaining the same results. Another interesting exercise is to try and see what the network has generalized, given the set of rules and the classification task learned.

Table 1
(MONKS 1) The number of input vectors queried, rules extracted, and rules remaining after simplification

MONKS 1	Input vectors	Queried	Extracted	Simplified
h_0	131072	18724	9455	2633
h_1	131072	18598	9385	536
h_2	131072	42776	21526	1793
o	8	8	2	1

Table 2
(MONKS 2) The number of input vectors queried, rules extracted, and rules remaining after simplification

MONKS 2	Input vectors	Queried	Extracted	Simplified
h_0	131072	131070	58317	18521
h_1	131072	43246	21769	5171
o	4	4	1	1

Table 3
(MONKS 3) The number of input vectors queried, rules extracted, and rules remaining after simplification

MONKS 3	Input vectors	Queried	Extracted	Simplified
h_0	131072	18780	9240	3311
h_1	131072	18618	9498	794
h_2	131072	43278	21282	3989
h_3	131072	18466	9544	1026
o	16	14	8	2

6.2. DNA sequence analysis

Molecular Biology is an area of increasing interest for the analysis and application of computational learning systems. Specifically, DNA sequence analysis problems have recently become a benchmark for the comparison of the performance of different learning methods. We apply the extraction algorithm on *Eukaryotes Promoter Recognition* and *Prokaryotes Splice Junction Determination*, which are very large real world problems. Differently from the Monk's Problems, now an exhaustive pedagogical extraction (sound and complete) turns out to be impossible due to the large number of input neurons: the networks trained in both problems contain more than 200 input neurons.

In what follows we briefly introduce the problems in question from a computational application perspective (see [37] for a proper treatment of the subject). A DNA molecule contains two strands that are linear sequences of nucleotides. The DNA is composed

Table 4
Set of rules extracted for the Monk's Problem 1

Rules for o
$\sim h_1 \sim h_2 \rightarrow o$
Rules for h_1
$\sim abcd \sim e \rightarrow h_1$
$bd \sim e \sim l \rightarrow h_1$
$b \sim i \sim lmn \rightarrow h_1$
$bcd(\sim l \vee \sim ef) \rightarrow h_1$
$b \sim ef(mn \vee mo) \rightarrow h_1$
$\sim abdf(\sim l \vee m \vee n) \rightarrow h_1$
$mno(\sim l \vee b \sim e \vee d \sim e \vee bc \vee cd \vee \sim ab \vee bf) \rightarrow h_1$
$1(mno) \wedge (bc \sim e \sim l \vee cd \sim e \sim l \vee \sim abcd \vee bcdf) \rightarrow h_1$
$1(mno) \wedge (bd \sim e \vee bd \sim l \vee b \sim ef \sim l \vee \sim ab \sim e \sim l) \rightarrow h_1$
Rules for h_2
$a \sim b \sim dek \sim l \rightarrow h_2$
$ac \sim dem \sim q \rightarrow h_2$
$a \sim b \sim def \sim l \rightarrow h_2$
$ae \sim gjm(n \vee o) \rightarrow h_2$
$\sim be \sim g \sim ln(a \vee \sim d) \rightarrow h_2$
$a \sim b \sim de \sim l(c \vee \sim h) \rightarrow h_2$
$\sim b \sim de \sim g \sim l(m \vee o) \rightarrow h_2$
$a \sim b \sim de \sim l(j \vee p \vee i) \rightarrow h_2$
$a \sim be \sim l \sim q(\sim d \vee m) \rightarrow h_2$
$a \sim be \sim g \sim l(\sim d \vee m \vee o) \rightarrow h_2$
$aem(\sim gn \sim p \vee \sim go \sim p \vee \sim hkn \vee \sim hko) \rightarrow h_2$
$1(mno) \wedge (\vee a \sim bc \sim d \sim l \vee \sim bc \sim de \sim l) \rightarrow h_2$
$1(mno) \wedge (ac \sim def \vee a \sim b \sim df \sim l \vee \sim b \sim def \sim l) \rightarrow h_2$
$1(mno) \wedge (a \sim bef \sim l \vee a \sim b \sim d \sim g \sim l \vee a \sim be \sim h \sim l) \rightarrow h_2$
$1(mno) \wedge (a \sim de \sim h \vee a \sim de \sim g \vee a \sim de \sim l \vee a \sim b \sim de) \rightarrow h_2$
$1(mno) \wedge (a \sim b \sim d \sim h \sim l \vee \sim b \sim de \sim h \sim l \vee a \sim bce \sim l) \rightarrow h_2$

from four different nucleotides—*adenine, guanine, thymine, and cytosine*—which are abbreviated by a, g, t, c, respectively. Some sequences of the DNA strand, called genes, serve as a blueprint for the synthesis of proteins. Interspersed among the genes are segments, called non-coding regions, that do not encode proteins.

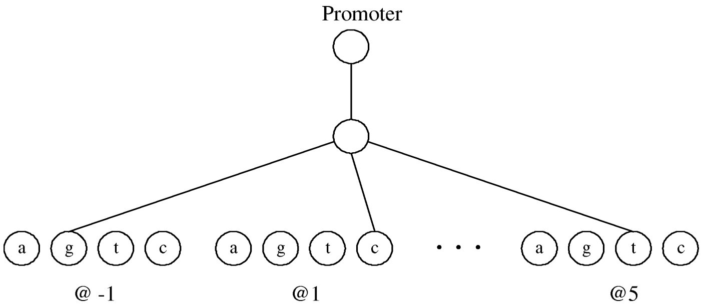


Fig. 22. Part of the network for Promoter Recognition.

Following [36], we use a special notation to identify the location of nucleotides in a DNA sequence. Each nucleotide is numbered with respect to a fixed, biologically meaningful, reference point. For example, "*@3 atcg*" states the location relative to the reference point in the DNA, followed by the sequence of symbols that must occur, i.e., an a must appear three nucleotides to the right of the reference point, followed by a t four nucleotides to the right of the reference point and so on. By convention, location zero is not used, and '$*$' indicates that any nucleotide will suffice in a particular location. Each location is encoded in the network by four input neurons, representing nucleotides a, g, t and c, in this order. Fig. 22 shows part of the network for Promoter Recognition. For example, suppose that input vectors with $@{-1}\ g = 1$, $@1\ c = 1$ and $@5\ t = 1$ activate the output Promoter. We want to extract a rule of the form $@{-1}\ gc * ** \ t \rightarrow \mathit{Promoter}$.

The first application is Prokaryotic [18] Promoter Recognition. Promoters are short DNA sequences that precede the beginning of genes. The aim of "*Promoter Recognition*" is to identify the starting location of genes in long sequences of DNA. The input layer of the network for this task contains 228 neurons (57 consecutive DNA nucleotides), its single hidden layer contains 16 neurons, and its output neuron is responsible for classifying the DNA sequence as promoter or nonpromoter. The set of training examples consists of 48 promoter and 48 nonpromoter DNA sequences, while the test set contains only 10 examples.

The second application is Eukaryotic [19] Splice-Junction Determination. Splice junctions are points on a DNA sequence at which the non-coding regions are removed during the process of protein synthesis. The aim of "*Splice Junction Determination*" is to recognize the boundaries between the part of the DNA retained after splice—called exons—and the part that is spliced out—the introns. The task consists, therefore, of recognizing exon/intron (E/I) boundaries and intron/exon (I/E) boundaries. Each example is a DNA sequence with 60 nucleotides (240 input neurons), where the center is the reference point. The network contains 26 neurons in its single hidden layer, while two output neurons are responsible for classifying the DNA sequences into E/I or I/E. The third category (neither E/I nor I/E) is considered true when neither output neurons are active. The training set for this task contains 1000 examples, in which approximately 25% are of I/E boundaries, 25% are of

[18] Prokaryotes are single-celled organisms that do not have a nucleus, e.g., E. Coli.
[19] Unlike prokaryotic cells, eukaryotic cells contain a nucleus, and so are higher up the evolutionary scale.

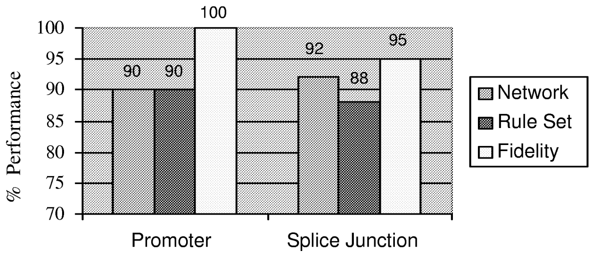


Fig. 23. The accuracy of the network, the accuracy of the rule set and the fidelity of the rule set to the network for the Promoter Recognition and Splice Junction Determination problems.

E/I boundaries and the remaining 50% are neither. We use a test set with 100 examples. Note that for the splice junction problem, we should not evaluate each output neuron individually. Instead, the combined activation of output neurons E/I and I/E should be considered.

In both applications, due to the intractability of the set of input vectors (2^{228} and 2^{240} elements each), we limit the maximum number of rules generated to 50,000 per hidden neuron. We also speed up the search process by doing the following: we jump, in a kind of binary search, from the ordering's minimum element to a new minimal element in the frontier at which input vectors start to generate rules. [20]

Fig. 23 displays the accuracy of the network, the accuracy of set of rules, and the fidelity of the set of rules to the network, for both the Promoter Recognition and Splice Junction Determination problems. The results reported were obtained using $A_{min} = 0.5$. In the promoter recognition task, the network classified 9 of the 10 test set examples correctly. The rule set extracted for this task classified the same 9 examples correctly, and thus the fidelity of the rule set to the network was 100%. In the splice junction problem, the network classified correctly 92 out of 100 examples. The rule set for this task classified 88 out of 100 examples correctly, and 7 of the 8 examples wrongly classified by the network were wrongly classified by the rule set. As a result, the fidelity of the rule set to the network was 95%.

The results obtained for the Promoter problem do not have statistical significance due to the reduced number of examples available for testing. However, the accuracy of the set of rules with respect to the network's training set was 90.6%, therefore similar to that obtained for the test set. Unfortunately, it is not easy to compare the results here obtained with the ones in [10,30,34]; differences in training and testing methodology are sufficient to preclude comparisons. For example, in [30] Setiono trains a network with three output neurons for the splice junction determination problem, while in [34] Towell uses cross-validation to test the network and the accuracy of the set of rules. To further complicate matters, the figures reported by Towell, concerning the results obtained by the MofN and Subset methods, refer to the training sets of the networks. Towell points out, though, that

[20] Instead of searching from the ordering's maximum and minimum elements, we pick an input vector at distance $n/2$ from them, where n is the number of input neurons, and query it. If it activates the output then it becomes a new maximal element; otherwise, it becomes a new minimal element. We carry on with this process until maximal and minimal elements are at distance 1 from each other.

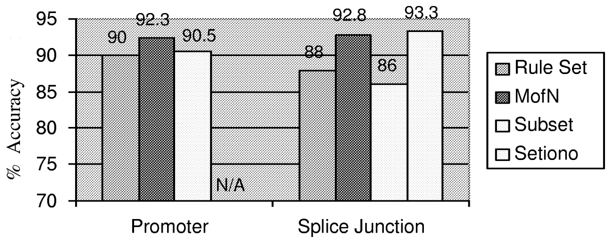


Fig. 24. Comparison with the accuracy obtained by other extraction methods in the Promoter Recognition and Splice Junction Determination problems.

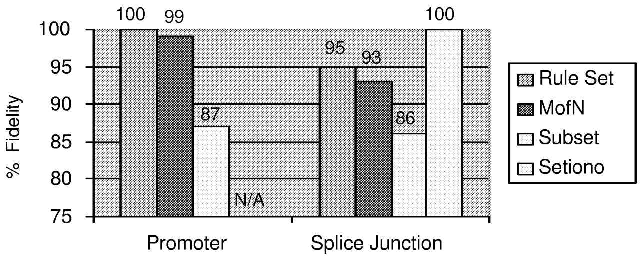


Fig. 25. Comparison with the fidelity achieved by other extraction methods in the Promoter Recognition and Splice Junction Determination problems.

the figures with respect to the test sets of the networks are similar. Finally, both Towell [34] and Fu [10] extract rules from networks in which a background knowledge had been inserted, while Setiono uses networks trained with no prior knowledge.

Nevertheless, in Fig. 24, we present the accuracy obtained by our extraction method, in comparison with MofN, Subset and Setiono's method, in both the Promoter and Splice Junction domains. The fidelity achieved by these extraction algorithms, again in the Promoter and Splice Junction domains, is shown in Fig. 25. In [30], 100% of fidelity (which we report here) seems to be assumed from the observation that the accuracy of the set of rules is identical to that of the network. However, this may not be the case when less than 100% of accuracy is achieved. In spite of the above mentioned differences in evaluation methodology, one can observe from Figs. 24 and 25 that, apart from the poor fidelity of the Subset method, our extraction method is within a margin of error of less than 5.5% from the results obtained by the remaining methods in both applications.

Finally, a comparison between the sizes of the sets of rules extracted by each of the above methods indicates that a drawback of our extraction algorithm lies in the much larger size of the set of rules, at least before the simplification of rules is carried out. On the other hand, the above experiments also show that an advantage of our method is the fact that a provably sound extraction is feasible even for very large networks. [21] We will address the

[21] We believe that the proof of soundness of the extraction algorithm is a prerequisite for the achievement of a good fidelity in any application.

problem of readability, and present some alternatives to counteract it, in the discussion at the end of this section.

6.3. Power systems fault diagnosis

Finally, we apply the extraction algorithm to power systems' fault diagnosis. Power systems' applications are an example of safety-critical domains, so that the soundness of the explanations provided by the set of rules extracted is of great importance. In this application, we can also illustrate the extraction of rules with classical negation ($\neg$), together with default negation ($\sim$), because some neurons are labelled $\neg x$ in the network's input and output layers (see [11] for the motivation behind adding classical negation to logic programs; [22] see [4] about encoding background knowledge with classical negation into neural networks).

Fig. 26 shows a simplified version of a real power plant. The system has two generators, two transformers with their respective circuit breakers, two buses (main and auxiliary) and two transmission lines also with their respective circuit breakers. Each transmission line has six associated alarms: breaker status (indicates whether it is open or not), phase over-current (shows that there was an over-current in the phase line), ground over-current (shows that there was an over-current in the ground line), timer (shows that there was a distant fault from the power plant generator), instantaneous (shows that there was a close-up fault from the power plant generator), and auxiliary (indicates that the transmission line is connected to the auxiliary bus). In addition, each transformer has three associated alarms: breaker status (indicates whether it is open or not), overloading (shows that there was a transformer overload) and auxiliary (indicates that the transformer is connected to the auxiliary bus). Finally, there are five alarms associated with the by-pass circuit breaker: breaker status, phase over-current, ground over-current, timer and instantaneous.

Certain combinations of the set of alarms indicate faults at Transmission Line 01 (11 possible kinds of faults), Transmission Line 02 (11 possible kinds of faults), or both (1 possible fault). In addition, each transformer may present three different kinds of faults. Finally, some alarms indicate the inexistence of a fault in the main bus or in each of the transformers (see [31] for details). We train a network with 23 input neurons, which represent the set of alarms of the power plant, 32 output neurons, which represent the set of faults of the power plant, and 35 hidden neurons in a single hidden layer. We do so using standard backpropagation. Each training example associates a set of alarms with possible faults. Some examples contain a unique fault associated with each set of alarms. Other examples associate many possible faults with each set of alarms. The set of 278 training examples contains approximately 10% of noise. [23] We use two test sets: one with 92 examples, in which only single faults are associated with each set of alarms, and another with 70 examples, in which multiple faults are associated with each set of alarms.

Figs. 27, 28 and 29 display the accuracy of the network, the accuracy of the rule set and the fidelity of the rule set to the network with respect to the test set with single faults,

[22] In this case, each concept of the network presents three possible values: *true*, *false* and *unknown*. In our application, either there is a fault (x), or there is not a fault ($\neg x$), or yet there is no evidence of a fault ($\sim x$).

[23] The absence of one or more of the alarms.

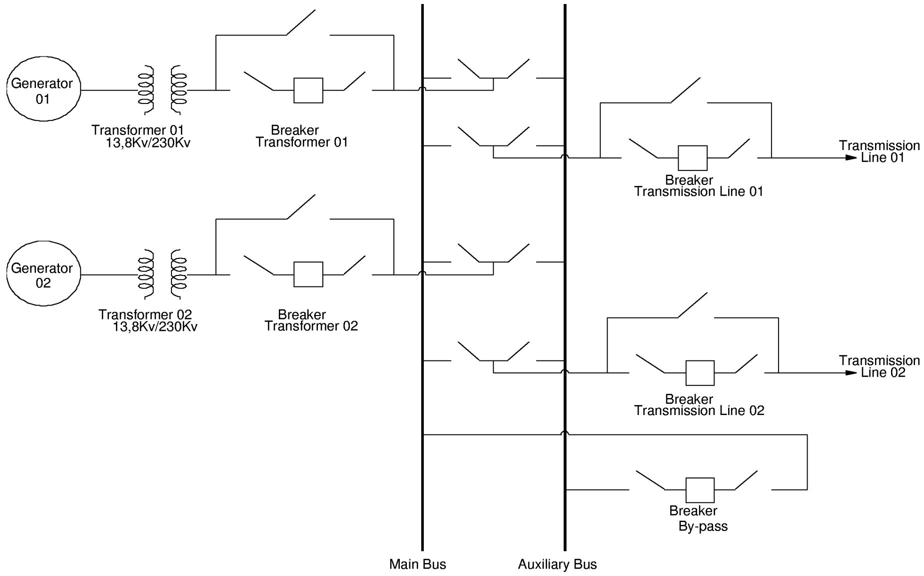


Fig. 26. Configuration of a simplified power system generation plant.

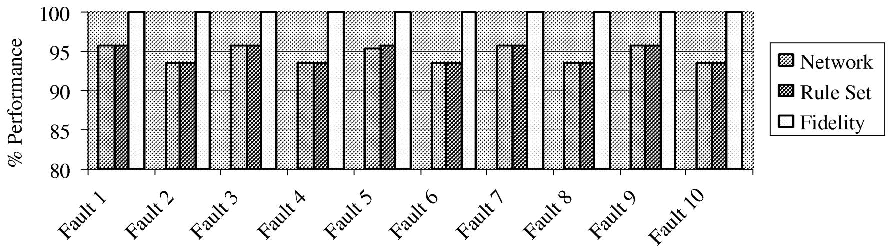


Fig. 27. Network, Rule Set and Fidelity percent with respect to the single faults test set (outputs 1–10).

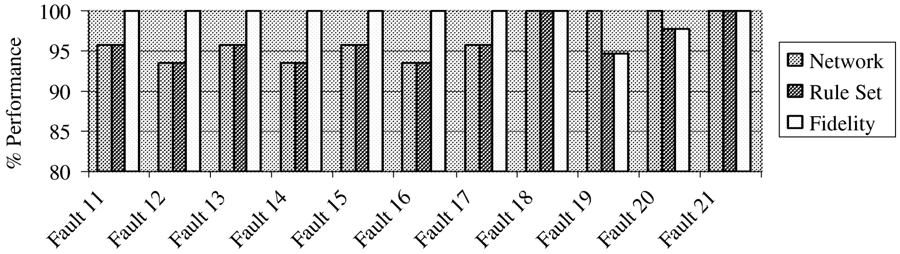


Fig. 28. Network, Rule Set and Fidelity percent with respect to the single faults test set (outputs 11–21).

for each output neuron. For example, for output neuron Fault 1 (Fig. 27), the network's accuracy was 95.7% (4 misclassifications in 92 examples), the accuracy of the set of rules extracted was also 95.7%, and the fidelity of the set of rules to the network was 100%,

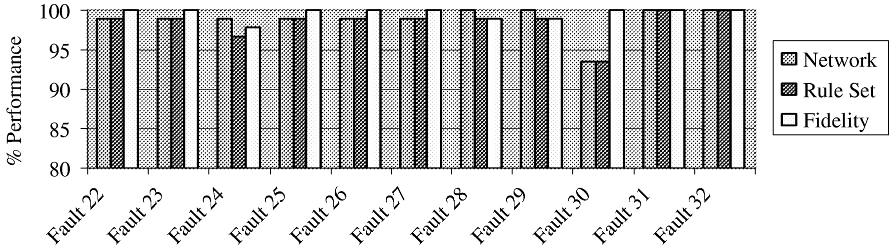


Fig. 29. Network, Rule Set and Fidelity percent with respect to the single faults test set (outputs 22–32).

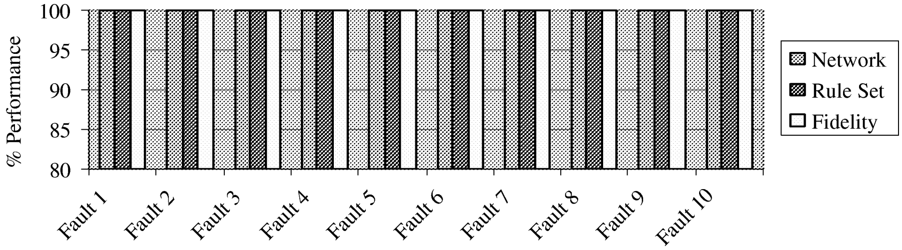


Fig. 30. Network, Rule Set and Fidelity percent with respect to the multiple faults test set (outputs 1–10).

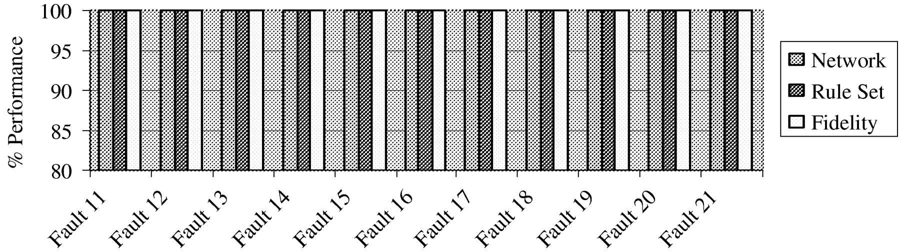


Fig. 31. Network, Rule Set and Fidelity percent with respect to the multiple faults test set (outputs 11–21).

i.e., the network and the set of rules misclassified the same 4 examples. Figs. 30, 31 and 32 show the same parameters for the test set with multiple faults. A typical rule extracted from the network for this problem is of the form:

$$\neg \mathit{Fault}(\mathit{Main\_Bus}, \ \mathit{Trans\_Line\_}01) \leftarrow \mathit{Alarm}\,(\mathit{Auxiliary\_Bus}, \ \mathit{Trans\_Line\_}01), \\ \sim\!\mathit{Alarm}\,(\mathit{Main\_Bus}, \ \mathit{Trans\_Line\_}01).$$

The results show the percentage of successful diagnosis achieved for each failure independently. Apart from Faults 24 and 30 in the multiple faults case, the accuracy of the rule set is very good. Similarly, the fidelity of the rule set to the network is excellent in most cases, and in general better than the accuracy of the rule set. Not surprisingly, this indicates that the extraction algorithm prioritizes fidelity over accuracy, i.e., it tries to mimic the network's behavior, which is a result of soundness.

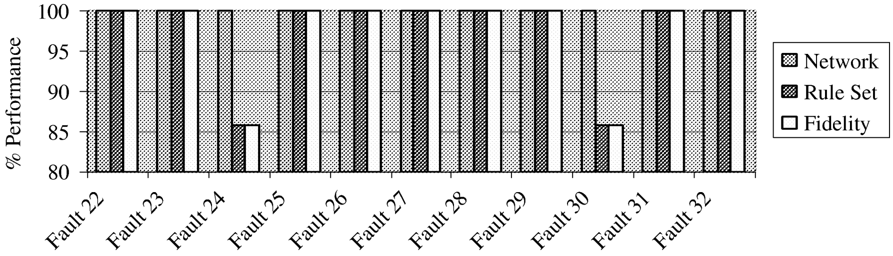


Fig. 32. Network, Rule Set and Fidelity percent with respect to the multiple faults test set (outputs 22–32).

However, the performance of systems of fault diagnosis is typically evaluated not only by determining the percentage of successful diagnosis, but also the average size of the ambiguity set (when the system isolates failures from several possible fault modes, but fails to correctly identify the set of faults). [24] For the network, the average size of the ambiguity set was 0.5% and 0% of the size of the set of activated faults, respectively, for the single and multiple faults test sets. For the rule set extracted, the size of the ambiguity set was 2.2% and the same 0% of the size of the set of activated faults, again for the single and multiple faults test sets.

6.4. Discussion

The above experimental results corroborate two important properties of the extraction system: it captures nonmonotonicity and it is sound. Nonmonotonicity is captured by the extraction of rules with default negation, as in the experiments on power systems fault diagnosis. Soundness is reflected in the very high fidelity achieved in the applications, by assuring that any rule extracted is actually encoded in the network, even if such a rule does not comply with the network's test set. The extraction system is, therefore, bound to produce a set of rules that tries to mimic the network, regardless of the network's performance in the training and test sets.

The above experiments also indicate that the drawback of the extraction system lies in the size of the set of rules. In comparison with [30] and [35], in the DNA sequence analysis domain, the number of rules extracted before any simplification is done is considerably bigger than, for example, the number of rules extracted by the *MofN* algorithm (despite the differences in syntax). It seems that less readability is the price one has to pay for soundness. The problem, however, is that we regard the proof of soundness as the minimum requirement of any method of rule extraction. We are, therefore, left with two possible courses of action:

(1) we can try to enhance readability by manipulating, e.g., simplifying, the extracted set of rules, or
(2) we can ignore the lack of readability of the set of rules as a whole, and concentrate on providing an explanation for each particular answer of the network.

[24] For each example, *size of ambiguity set* = (*number of wrongly activated outputs / number of activated outputs*) $\times$ 100.

As far as course of action (1) is concerned, there are many possible improvements to be made in our extraction system.

- Firstly, *M of N* simplifications (not yet implemented) can be very powerful, as in [35], in helping reduce the size of the rule set. Even better, simplifications could be made on the fly, at the same time that rules are generated.[25]
  In Section 4.2, we have seen an example of the relation between the ordering on the set of input vectors ($\boldsymbol{I}$) of a network and *M of N* rules. In fact, each valid *M of N* rule is associated with a valid subset of $\boldsymbol{I}$. For example, let $\boldsymbol{i}_1 = (-1, -1)$, $\boldsymbol{i}_2 = (-1, 1)$, $\boldsymbol{i}_3 = (1, -1)$ and $\boldsymbol{i}_4 = (1, 1)$. Let $\boldsymbol{I} = \{\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{i}_3, \boldsymbol{i}_4\}$ and $\sup(\boldsymbol{I}) = (1, 1)$. There are 5 valid subsets of $\boldsymbol{I}$, apart from $\emptyset$, namely, $\{\boldsymbol{i}_4\}$, $\{\boldsymbol{i}_4, \boldsymbol{i}_3\}$, $\{\boldsymbol{i}_4, \boldsymbol{i}_2\}$, $\{\boldsymbol{i}_4, \boldsymbol{i}_3, \boldsymbol{i}_2\}$, and $\boldsymbol{I}$ itself. If $(1, 1) = [a, b]$ then each of these subsets correspond, respectively, to the following *M of N* rules: $2(a, b)$, $1(a)$, $1(b)$, $1(a, b)$, and $0(a, b)$. Any other *M of N* rule is not valid due to the ordering $\preceq$ on $\boldsymbol{I}$. For example, $1(a, \sim b)$ would require the set $\{\boldsymbol{i}_4, \boldsymbol{i}_2, \boldsymbol{i}_1\}$ to be also a valid subset of $\boldsymbol{I}$, but this is impossible according to $\preceq$. *M of N* rule $1(a, \sim b)$ would require $\sup(\boldsymbol{I}) = (1, 1) = [a, \sim b]$, in which case rule $1(a, b)$ would not be a valid *M of N* rule, for the same reason as described above.[26]
  The relation between *M of N* rules and subsets of $\boldsymbol{I}$ could facilitate the extraction of more compact sets of rules, thus improving readability. By manipulating *M of N* rules, as in [23], a neater set of rules could also be derived. The characterization of an algebra for manipulating *M of N* rules is work in progress.
- Improvements could also be made in the optimization of the system's search process, exploring the ordering on the set of input vectors, and adding some new heuristics to the extraction algorithm. An example is what we have done in the DNA sequence analysis case, when we jump to new minimal elements in the ordering.
  The efficiency of the search process could also be enhanced by the implementation of a *time-slice* for each output neuron. This would help the extraction not to get stuck in the generation of thousands of rules about an output, while no rule about the remaining outputs is created. As far as efficiency is concerned, a parallel implementation of the extraction system would be the ultimate goal.
- Finally, a possible extension of the extraction algorithm concerns the extraction of *metalevel priorities* [24,25] directly from the network's *Hidden-to-Output BNS*s. Negative weights from hidden to output neurons implement a preference relation (see [6]). We could use this information to extract directly from the network, together with object level rules, a set of metalevel priorities between rules. Alternatively, this could be done after the extraction, when the rules are assembled to derive the final rule set. The result would be the enhancement of readability, by means of the use of a more compact representation.
  Consider, for example, a non-regular network N from which the following set of rules is extracted $R = \{ab \to h_1,\ c \to h_2,\ h_1 \to \neg x,\ \sim h_1 h_2 \to x\}$. When hidden

[25] The idea here is to implement a *buffer* of rules extracted and, whenever a new rule is generated, try to simplify it together with the rules in the buffer. Potentially good rules for simplification, the ones with many '*don't cares*', would remain in the buffer for longer periods.

[26] In fact, this is the reason why *M of N* rules ought to be seen as simplifications.

neurons h_1 and h_2 are eliminated, we obtain $R' = \{ab \to \neg x,\ \sim ac \to x,\ \sim bc \to x\}$. However, by associating h_1 with $r_1 : ab \to \neg x$ and h_2 with $r_2 : c \to x$, we find out that R' is equivalent to $R'' = \{r_1 : ab \to \neg x, r_2 : c \to x\}$ together with the preference relation $r_1 \succ r_2$, which should read "rule r_1 has priority over rule r_2". Clearly, R'' is more readable than R'.

The idea behind course of action (2) is to provide an explanation for individual answers of the network, instead of trying to understand what is computed by it as a whole. When we extract rules from a trained network, we obtain a database, which can be used instead of the network. By querying the database with a particular answer of the network, using, for instance, an automatic theorem prover, we may use the steps of the proof of a literal to provide a symbolic explanation for such an answer of the network. Note that this explanation will only be reliable if the extraction of rules is sound. Of course, when one takes course of action (2), some interesting features of the network might never be found. On the other hand, in this case, even very large sets of rules are not a major concern.

7. Conclusion

We have seen that most decompositional methods of extraction are unsound. On the other hand, sound and complete pedagogical extraction methods have exponential complexity. We call this problem the *complexity* $\times$ *quality* trade-off. In order to ameliorate it, we started by analyzing the cases where regularities can be found in the set of weights of a neural network. If such regularities are present, a number of *pruning rules* can be used to safely reduce the search space of the extraction algorithm. These pruning rules reduce the extraction algorithm's complexity in some interesting cases. Notwithstanding, we have shown that the extraction method is sound and complete with respect to an exhaustive pedagogical extraction. A number of *simplification rules*, that fit very well into the extraction method due to a counterpart graphical representation on the network's input vectors' ordering, also help reducing the length of the extracted set of rules.

We then extended the extraction algorithm to the cases where regularities are not present in the network as a whole. That is the general case, since we do not fix any constraints on the network's learning algorithm. However, we have identified subnetworks of non-regular networks that always contain regularities, by showing that the network's building block, here called Basic Neural Structure (*BNS*), is regular. As a result, using the same underlying ideas, we were able to derive rules from each *BNS*. In this case, however, we were applying a decompositional approach, and our problem was how to assemble the final rule set of the network. We needed to provide a special treatment for *Hidden-to-Output BNS*s, since the activation values of hidden neurons are not discrete, but real numbers in the interval $(-1, 1)$. In order to deal with that, we assumed, without loss of generality, two possible intervals of activation $(-1, A_{max})$ and $(A_{min}, 1)$, and performed a worst case analysis. Finally, we used the completeness of the extraction from *Input-to-Hidden BNS*s to assemble the final set of rules of the network, and show that the general case extraction method is still sound.

In this paper, we have investigated the problem of extracting the symbolic knowledge encoded in trained neural networks. Although neural networks have shown very good performance in many application domains, one of their main drawbacks lies in the incapacity to explain the reasoning mechanisms that justify a given answer. This motivated the first attempts towards extracting a symbolic knowledge from trained networks, dating back to the end of the 1980s. The problem of knowledge extraction turned out to be one of the most interesting open problems in the field. So far, some extraction algorithms were proposed [1,3,9,10,26,30,35] and had their effectiveness empirically confirmed using certain applications as benchmark. Some theoretical results have also been obtained [5,10, 15,33]. However, we are not aware of any extraction method that fulfils the following list of desirable properties suggested by Thrun in [33]:

(1) no architectural requirements;
(2) no training requirements;
(3) correctness; and
(4) high expressive power.

The extraction algorithm presented here satisfies the above requirements (2) and (3). It does impose, however, some restriction on the network's architecture. For instance, it assumes that the network contains a single hidden layer. This, according to the results of Hornik et al. [16], is not a drawback though. In what concerns the expressive power of the extracted set of rules, our extraction algorithm enriches the language commonly used by adding default negation. This is done because neural networks encode nonmonotonicity. In spite of that, we believe that item (4) is the subject, among the above, that needs most attention and further development.

As future work, we would like to tackle the problem of rule extraction from networks with continuous inputs. Clearly, when the Translation Algorithm of CIL^2P is used (Fig. 1, step (1)), one can convert numerical attributes into discrete ones, using any desired degree of accuracy, as done in [30], for example. In this case, the extraction algorithm of CIL^2P can be applied directly, without any modifications. The interesting case for future investigation arises when a network trained with continuous inputs is simply given, and we want to extract rules from it, i.e., instead of the whole system, we can only use the extraction module of CIL^2P. In this case, it seems that the process used for the extraction of rules from hidden to output subnetworks should be applied also for input to hidden subnetworks. As before, the ordering on the input vectors of regular (sub)networks is valid for any activation values chosen. The problem, though, lies in the choice of "good" activation values. It is similar to the problem of defining a fuzzification scheme and its membership functions, as shown in [20]. The proof of soundness in this case, however, seems to be a big challenge, and, in our point of view, soundness should be regarded as the minimum requirement of any rule extraction method.

In addition, the extension of the extraction system to perform a stochastic search, as opposed to a deterministic search, in the lattice of input vectors seems promising. Stochastic searches have outperformed deterministic ones in a variety of logic and AI tasks, starting with the work of Selman, Levesque and Mitchell on satisfiability [28]. Consequently, we believe that a stochastic search of the frontier of activations in the lattice of input vectors could improve the experimental results obtained with the current (deterministic) implementation of the extraction algorithm.

Acknowledgements

We are grateful to Gerson Zaverucha, Valmir Barbosa, Luis Alfredo Carvalho, Steffen Hoelldobler, Franz Kurfess and Luis Lamb for useful discussions. We would especially like to thank Alberto de Souza, Stefan Rueger and the anonymous referees for their comments. The first author was partially supported by the Brazilian Research Agency CAPES.

References

[1] J.A. Alexander, M.C. Mozer, Template-based algorithms for connectionist rule extraction, in: G. Tesauro, D. Touretzky, T. Leen (Eds.), Advances in Neural Information Processing Systems (NIPS), Vol. 7, MIT Press, Cambridge, MA, 1995.

[2] R. Andrews, J. Diederich, A.B. Tickle, A survey and critique of techniques for extracting rules from trained artificial neural networks, Knowledge-Based Systems 8 (6) (1995).

[3] R. Andrews, S. Geva, Inserting and extracting knowledge from constrained error backpropagation networks, in: Proc. 6th Australian Conference on Neural Networks, 1995.

[4] A.S. d'Avila Garcez, G. Zaverucha, V.N.L. da Silva, Applying the connectionist inductive learning and logic programming system to power system diagnosis, in: Proc. IEEE International Joint Conference on Neural Networks, ICNN97, Houston, TX, 1997.

[5] A.S. d'Avila Garcez, G. Zaverucha, The connectionist inductive learning and logic programming system, Applied Intelligence Journal (Special Issue on Neural Networks and Structured Knowledge) 11 (1) (1999) 59–77.

[6] A.S. d'Avila Garcez, K. Broda, D.M. Gabbay, Metalevel priorities and neural networks, in: Proc. Workshop on the Foundations of Connectionist-Symbolic Integration, ECAI 2000, Berlin, Germany, 2000.

[7] N.K. Bose, P. Liang, Neural Networks Fundamentals with Graphs, Algorithms, and Applications, McGraw-Hill, New York, 1996.

[8] M.W. Craven, J.W. Shavlik, Using sampling and queries to extract rules from trained neural networks, in: Proc. 11th International Conference on Machine Learning, New Brunswick, NJ, 1994.

[9] M.W. Craven, Extracting Comprehensible Models from Trained Neural Networks, University of Wisconsin, Madison, WI, 1996.

[10] L. Fu, Neural Networks in Computer Intelligence, McGraw-Hill, New York, 1994.

[11] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Computing 9 (1991) 365–386.

[12] S.I. Gallant, Neural Networks Learning and Expert Systems, MIT Press, Cambridge, MA, 1993.

[13] J. Hertz, A. Krogh, R.G. Palmer, Introduction to the Theory of Neural Computation, Santa Fe Institute, Studies in the Science of Complexity, Addison-Wesley, Reading, MA, 1991.

[14] M. Hilario, An overview of strategies for neurosymbolic integration, in: Connectionist-Symbolic Integration: From Unified to Hybrid Approaches—IJCAI-95, Montreal, Que., 1995.

[15] S. Holldobler, Y. Kalinke, Toward a new massively parallel computational model for logic programming, in: Proc. Workshop on Combining Symbolic and Connectionist Processing, ECAI-94, Amsterdam, 1994.

[16] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (1989) 359–366.

[17] H. Kautz, M. Kearns, B. Selman, Horn approximations of empirical data, Artificial Intelligence 74 (1995) 129–145.

[18] N. Lavrac, S. Dzeroski, Inductive Logic Programming: Techniques and Applications, Ellis Horwood Series in Artificial Intelligence, Ellis Horwood, Chichester, 1994.

[19] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1987.

[20] A. Lozowski, J.M. Zurada, Extraction of linguistic rules from data via neural networks and fuzzy approximation, in: I. Cloete, J.M. Zurada (Eds.), Knowledge-Based Neurocomputing, MIT Press, Cambridge, MA, 2000 , pp. 403–417.

[21] W. Marek, M. Truszczynski, Nonmonotonic Logic: Context Dependent Reasoning, Springer, Berlin, 1993.

[22] M. Minsky, Logical versus analogical, symbolic versus connectionist, neat versus scruffy, AI Magazine 12 (2) (1991).
[23] F. Maire, A partial order for the M-of-N rule extraction algorithm, IEEE Transactions on Neural Networks 8 (6) (1997) 1542–1544.
[24] D. Nute, Defeasible logic, in: D. Gabbay, C.J. Hogger, J.A. Robinson (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 3, Oxford Science Publications, Oxford, 1994, pp. 353–396.
[25] H. Prakken, G. Sartor, Argument-based extended logic programming with defeasible priorities, J. Appl. Non-Classical Logic 7 (1/2) (1997) 25–75.
[26] E. Pop, R. Hayward, J. Diederich, RULENEG: Extracting Rules from a Trained ANN by Stepwise Negation, QUT NRC, 1994.
[27] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation, in: D.E. Rumelhart, J.L. McClelland and the PDP Research Group (Eds.), Parallel Distributed Processing, Vol. 1, MIT Press, Cambridge, MA, 1986.
[28] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proc. AAAI-92, San Jose, CA, 1992.
[29] R. Setiono, A penalty-function for pruning feedforward neural networks, Neural Computation 9 (1997) 185–204.
[30] R. Setiono, Extracting rules from neural networks by pruning and hidden-unit splitting, Neural Computation 9 (1997) 205–225.
[31] V.N. Silva, G. Zaverucha, G. Souza, An integration of neural networks and nonmonotonic reasoning for power systems diagnosis, in: Proc. IEEE International Conference on Neural Networks, ICNN-95, Perth, Australia, 1995.
[32] S.B. Thrun et al., The MONK's problem: A performance comparison of different learning algorithms, Technical Report, CMU-CS-91-197, Carnegie Mellon University, Pittsburgh, PA, 1991.
[33] S.B. Thrun, Extracting provably correct rules from artificial neural networks, Technical Report, Institut für Informatik, Universität Bonn, Bonn, 1994.
[34] G.G. Towell, Symbolic Knowledge and Neural Networks: Insertion, Refinement and Extraction, Ph.D. Thesis, University of Wisconsin, Madison, WI, 1992.
[35] G.G. Towell, J.W. Shavlik, The extraction of refined rules from knowledge based neural networks, Machine Learning 13 (1993).
[36] G.G. Towell, J.W. Shavlik, Knowledge-based artificial neural networks, Artificial Intelligence 70 (1994) 119–165.
[37] J.D. Watson, N.H. Hopkins, J.W. Roberts, J.A. Steitz, A.M. Weiner, Molecular Biology of the Gene, Vol. 1, Benjamin Cummings, Menlo Park, CA, 1987.