

# Extracting Rules from Neural Networks as Decision Diagrams

Jan Chorowski, Jacek M. Zurada, *Fellow, IEEE*

**Abstract**—Rule extraction from neural networks solves two fundamental problems: it gives insight into the logic behind the network and, in many cases, it improves the network’s ability to generalize the acquired knowledge. This article presents a novel, eclectic approach to rule extraction from neural networks, named LORE, suited for multilayer perceptron networks with discrete (logical or categorical) inputs. The extracted rules mimic network behavior on training set and relax this condition on the remaining input space.

First, a multilayer perceptron network is trained under standard regime. It is then transformed into an equivalent form, returning the same numerical result as the original network, yet being able to produce rules generalizing the network output for cases similar to a given input. The partial rules extracted for every training set sample are then merged to form a decision diagram from which logic rules can be extracted.

A rule format explicitly separating subsets of inputs for which an answer is known from those with an undetermined answer is presented. A special data structure, the decision diagram, allowing efficient partial rule merging is introduced. With regard to rules’ complexity and generalization abilities, LORE gives results comparable with those reported previously. An algorithm transforming decision diagrams into interpretable boolean expressions is described. Experimental running times of rule extraction are proportional to the network’s training time.

**Index Terms**—Feedforward neural networks, rule extraction, logic rules, true false unknown logic, decision diagrams.

## I. INTRODUCTION

NEURAL networks are commonly used to solve many practical tasks, formulated typically as classification or regression problems. Their most attractive properties include good generalization capabilities, noise robustness, or the possibility of re-adapting a network after the initial training. However, the learned knowledge is most often hidden from the user and, for most practical purposes, neural networks are no more than black boxes.

Rule extraction from neural networks attempts to solve this drawback by providing a description of the inner workings of the network which, ideally, will retain high fidelity (produce similar answers to those of the network), while being easy to understand by humans. It has been an extensively studied research topic and the detailed surveys by Andrews et al. [1] and by Tickle et al. [2] provide a taxonomy of rule extraction methods and present a broad and detailed view on the subject. This paper will concentrate on a class of rule extraction

scenarios using feedforward perceptron networks with discrete inputs.

One of the main criteria in describing rule extraction methods is how the algorithm makes use of the existing neural network. In the *pedagogical* approach, the internal structure of the network (or any other classifier) is irrelevant and only the relation between the input and output is evaluated. For example, the method proposed by Craven and Shavlik named TREPAN [3] treats the network as an oracle used to statistically verify the correctness and significance of generated rules. In a similar way the method OSRE [4] finds for each training sample important inputs and uses them to form conjunctive rules. In contrast, the *decompositional* methods try to derive rules from the structure of the network. For example, Fu’s SUBSET method [5] finds for each neuron in the network subsets of its inputs causing the neuron to become active. An optimization minimizing the search space by sorting weights has been proposed by Krishnan [6]. Some decompositional methods rely on special network training. The Re-RX [7] method recursively trains, prunes and analyses a neural network to generate a set of hierarchical rules. Towell et al. described a very interesting method nicknamed KBANN used to refine existing rules [8]. Its main idea is to encode existing domain knowledge inside the network structure, then train such a specially initialized network, and finally extract new, better rules. Its successor, TOPGEN, is able to add new rules to a given rule set [9].

Rule extraction techniques which do not fall clearly into one of the above categories are called *eclectic*. An example, taken from the realm of support vector machines [10], uses the inner architecture of the trained SVM (computed support vectors, but reclassified using the SVM) to induce a decision tree.

Criteria commonly used to assess the usefulness of rule extraction methods are: *accuracy* – it measures the ability of the rules to properly classify previously unseen data (generalization ability), *fidelity* – it reflects how well the rules mimic the network, *consistency* – it describes how the rules differ between different training sessions, *comprehensibility* – it states how easy to understand a set of rules is by measuring the number of rules and their antecedents, and finally *computational complexity* – which reflects the needs of the process of rule generation.

It can be argued whether the rule extraction process should emphasize better agreement with the data set (better accuracy), or rather with the network (better fidelity) [11]. This paper reflects the opinion that mimicking the network behavior on

J. Chorowski and J. M. Zurada are with the Department of Computer and Electrical Engineering, University of Louisville, Louisville, KY 40208 USA. Manuscript received xxxxxxxx; revised xxxxxxxxxx.

the training samples, but relaxing this accuracy from it in the rest of the input space, might be a good solution.

One of the limitations of known rule extraction methods is their computational complexity. In most cases, the procedure performs an exponentially growing number of operations with regard to the size of the input vector or the size of the hidden layer. It follows, however, that only trivial cases can be solved by such methods.

In Appendix A we demonstrate two proofs showing that rule extraction algorithms aiming for perfect network fidelity are, in the general case, NP-hard. Decompositional methods can effectively solve some simple problems, such as listing the M-of-N rules describing a single perceptron (in a way similar to the KBANN method). However, merging rules extracted from hidden units into a global formula is computationally expensive.<sup>1</sup> Pedagogical methods must require at least as many operations as the decompositional ones, since they simply don't use all the information provided to produce the same result.

To make the rule extraction problem tractable, authors usually impose certain conditions limiting the search space, such as setting the maximum number of rule antecedents. This poses an important question to what extent such incomplete rules properly represent the network.

Our proposition is to limit the analysis to the part of the feature space close to the training set samples and narrow down the rule extraction problem to *provide a description of the function realized by the neural network near the samples from the training set*. Several arguments speak for this approach. First, the ultimate goal is to learn from the data, not from the network. Second, the network finds some relations in the training set. The further we diverge from it, the more complicated the decision surface of the network might be and the more unnecessary it becomes to faithfully describe it. Third, if one ever had to extract the rules from a network similar to the pathological ones used to prove NP-hardness of rule extraction, it must have come as the result of training. If the network learned such a complicated function, the relation must have been in the data first.

## II. DETAILED DESCRIPTION OF THE PROPOSED METHOD

The outline of LORE (LOcal Rule Extraction) is presented in Fig. 1. Its major components are a way of capturing network's actions for a given sample into a partial rule, an efficient data structure to manipulate the rules able to perform rule merging operation, and optionally, generalization and pruning.

To simplify the description of LORE this discussion is limited to the case of two classes (denoted by  $\mathcal{T}$  and  $\mathcal{F}$ ) computed by a network having only discrete inputs and only one hidden layer. The algorithms are easily extended to handle multi-class problems, as well as more complicated network structures. However, no continuous attributes are allowed.

<sup>1</sup>This problem is nicely dealt with in the KBANN algorithm – since hidden neurons have an associated meaning, there is no need to combine partial rules describing them. In the general case, however, hidden neurons show no understandable meaning and partial rules have to be merged if they are to be understood by a human.

```

fun LORE()
   $G \leftarrow \mathcal{U}$  {start with the empty rule}
  for all  $X \in$  Training Set do
    {Create new partial rule describing just this sample}
     $PR \leftarrow \text{derivePR}(\text{net}, X)$ 
     $G \leftarrow \text{merge}(G, PR)$ 
  end for {now  $G$  correctly classifies all training samples and
  needs to be extended over whole feature space  $\mathbb{I}$ }
   $G \leftarrow \text{generalize}(G)$ 
   $Pruned \leftarrow \text{prune}(G)$  {optionally further simplify}

```

Fig. 1. Outline of the LORE method

The notation is as follows: let the network have  $k$  input features and let  $F_i$  denote the  $i$ -th feature, taking  $n_i$  different values. When there is no confusion we will also use  $F_i$  to denote the set of values taken by the  $i$ -th feature. The space of network inputs, or the feature space, is thus  $\mathbb{I} = F_1 \times F_2 \times \dots \times F_k$ . For every feature  $F_i$ , the network has  $n_i$  inputs that use the 1-of- $N$  encoding. All network inputs take either the value 0 or 1.<sup>2</sup> For a feature  $F$  and a vector  $X$ , let  $X_F$  denote the part of  $X$  associated to that feature (the input values of encoded feature  $F$ , the weights connecting such inputs, etc.). Let  $K = \sum_{i=1}^k n_i$  be the total number of inputs. The network realizes a numerical function  $\bar{f} : \mathbb{R}^K \rightarrow \mathbb{R}$ . However, the only *valid* (meaningful) inputs are binary vectors of length  $K$  in which for every feature there is exactly one input having value 1 and all the others have value 0. Consequently, we will say that an input is invalid when it is not valid (i.e. for at least one feature all the inputs are zeroed, or more than one input is active).  $\bar{X}$  is to denote a real vector which is a valid network input corresponding to an input sample  $X \in \mathbb{I}$ . For those valid vectors we will introduce the logical function  $f : \mathbb{I} \rightarrow \{\mathcal{F}, \mathcal{T}\}$  defined in the usual way as:

$$f(X) = \begin{cases} \mathcal{F} & \text{if } \bar{f}(\bar{X}) < 0, \\ \mathcal{T} & \text{if } \bar{f}(\bar{X}) \geq 0. \end{cases} \quad (1)$$

The section below describes the important aspects of the algorithm.

### A. Classifying as True, False or Unknown to keep track of where a rule is applicable

During the execution of the algorithm partial rules are deduced. They are functions assigning to samples either the class label, or the special value  $\mathcal{U}$  (unknown) if the rule doesn't classify the sample.

**Definition 1.** A partial rule  $PR$  is a function  $PR : \mathbb{I} \rightarrow \{\mathcal{F}, \mathcal{T}, \mathcal{U}\}$  from the feature space into the set of classes augmented with the special value unknown, denoted as  $\mathcal{U}$ . Moreover we will define the domain of a partial rule as the subspace of valid inputs for which the partial rule's value is known:

$$\text{Dom}(PR) = \{X \in \mathbb{I} \mid PR(X) \neq \mathcal{U}\}$$

<sup>2</sup>For training purposes more suitable values would be  $\pm 1$ . The weights can be linearly scaled to accommodate a different input encoding after training.

Hence while we can apply a rule to all samples, only those belonging to that rule's domain will be correctly classified.

We will say that two rules agree (denoted by  $\approx$ ) if they identically classify samples belonging to the intersection of their domains:

**Definition 2.** Two partial rules  $R1$  and  $R2$  agree with each other if and only if:

$$\forall X \in \mathbb{I} X \in \text{Dom}(R1) \wedge X \in \text{Dom}(R2) \Rightarrow R1(X) = R2(X)$$

Two partial rules that agree can be merged into a new one whose value will be known on the union of their domains and agreeing with both of them.

**Definition 3.** The operator  $\oplus$  merges two partial rules that agree,  $R1$  and  $R2$ , such that:  $R = R1 \oplus R2$  if and only if:

$$\begin{aligned} R1 &\approx R2 \\ \text{Dom}(R) &= \text{Dom}(R1) \cup \text{Dom}(R2) \\ R &\approx R1 \text{ and } R \approx R2 \end{aligned}$$

Obviously, the network function  $f$  as given in (1) is a partial rule whose domain covers the entire feature space. The constant function  $U(X) = \mathcal{U}$  has an empty domain and is not useful for classification, but it agrees with all partial rules.

The algorithm presented in Fig. 1 starts with a partial rule  $G = \mathcal{U}$ . Then in a loop over each training sample the domain of  $G$  is extended to cover the processed sample and a part of its neighborhood. It can be proved that  $G$  agrees at every step with the network function and that its domain contains all the processed training samples. This means that whenever the value of  $G$  at  $X$  is not  $\mathcal{U}$ , it is equal to the network function value, i.e.  $G(X) \neq \mathcal{U} \Rightarrow G(X) = f(X)$ . However, if  $G(X) = \mathcal{U}$ , then the rule  $G$  doesn't provide any information about the class of  $X$ .

Upon the completion of the loop over the training set,  $G$  holds a partial rule which classifies every training sample and generalizes over some part of the input space  $\mathbb{I}$  in a similar way to the network. Next, to extend the partial rule's domain to the full feature space, a generalization procedure can be applied. Please note that this step breaks compatibility with the network and the generalized  $G$  does no longer agree with  $f$ , the network function (the rule set and network may disagree on previously unseen samples). Generalization can be followed by an optional simplification (pruning) step.

### B. Estimating minimal and maximal network excitation.

To derive a partial rule from each training sample  $S$  we will determine a subset of features that are sufficient to classify this sample. These will be called the *important* features of  $S$ . We will estimate network's output when the value of some features is not set and try to greedily deselect features whose elimination doesn't change the network's classification. The new partial rule used to classify a previously unseen sample  $X$  derived from a training sample  $S$  is then:

$$PR_S(X) = \begin{cases} \text{Class}(S) & \text{if } \forall F \in \text{Important}(S) S_F = X_F \\ \mathcal{U} & \text{otherwise.} \end{cases} \quad (2)$$

Meaning that if a sample  $X$  and a known training sample  $S$  have exactly the same value of all important features of  $S$ , then  $X$  is classified in the same way as  $S$ . Otherwise, the class of  $X$  is unknown to the rule and  $\mathcal{U}$  is returned.

Before delving into the case of a whole network, we will analyze just a single neuron.

1) *A single neuron:* Suppose we analyze a single neuron  $N$  having activation function:

$$N(X) = s(X \cdot W - b) = s\left(\sum_{F=1}^k X_F \cdot W_F - b\right) \quad (3)$$

where  $s(\cdot)$  is a sigmoidal transfer function,  $X \in \{0, 1\}^K$  is a valid input vector,  $W$  is the weight vector and  $b$  is the bias.

Let  $F$  be a selected feature. Consider the neuron  $N'$  with weights  $W'$  and bias  $b'$  obtained from  $N$  by subtracting from weights associated with the feature  $F$  and the bias the minimum weight for that feature, i.e.

$$W'_i = \begin{cases} W_F - \min(W_F) & \text{for } i = F \\ W_i & \text{for } i \neq F \end{cases} \quad (4a)$$

$$b' = b - \min(W_F). \quad (4b)$$

We will show that for all valid inputs (those having the property that for every feature exactly one of its associated inputs is 1, while the others are 0) the excitation of  $N'$  is equal to that of  $N$ , hence the two neurons are equivalent. However,  $N'$  can be used to calculate the minimum excitation of  $N$  if we don't specify the value of feature  $F$  by calculating its excitation for an input vector  $X'$  with all inputs associated with  $F$  zeroed.

**Theorem 1.** For any valid input  $X$ , the neuron  $N'$  obtained from a neuron  $N$  using the transformation (4) has exactly the same activation value. Furthermore for an input vector  $X'$  obtained from  $X$  by zeroing inputs associated with the feature  $F$ , i.e.  $X'_F = 0$ ,  $X'_i = X_i$  for  $i \neq F$  the neuron  $N'$  returns the minimum activation of  $N$  for all possible values of  $F$ .

*Proof:* Without loss of generality we can reorder the features, so that the feature  $F$  is the first feature. If the input vector  $X$  is valid, then exactly one input associated with the feature is set to 1, while the others are zeroed, i.e.  $X_1$  has exactly one 1 on the  $J$ -th position. Thus exactly one of  $W_1$ , weights associated with the first feature, is included in the summation. Then

$$\begin{aligned} X \cdot W - b &= \sum_{j=1}^{n_1} x_{1,j} w_{1,j} - b + \sum_{i=2}^k X_i \cdot W_i = \\ &= w_{1,J} - b + \sum_{i=2}^k X_i \cdot W_i = \\ &= (w_{1,J} - \min(W_1)) - (b - \min(W_1)) \quad (5) \\ &+ \sum_{i=2}^k X_i \cdot W_i = \\ &= \sum_{i=1}^k X_i \cdot W'_i - b' = X' \cdot W - b', \end{aligned}$$

since the only nonzero element in  $X_1$  is  $x_{1,J} = 1$ .

To prove the second part observe that:

$$\begin{aligned} \min_{X_1} N(X) &= \min_{X_1} s \left( X_1 \cdot W_1 + \sum_{i=2}^k X_i \cdot W_i - b \right) = \\ &= s \left( \min(W_1) + \sum_{i=2}^k X_i \cdot W_i - b \right) = \\ &= s \left( \sum_{i=2}^k X_i \cdot W_i - (b - \min(W_1)) \right) = \\ &= N'(X') \end{aligned} \quad (6)$$

By repeatedly applying the transformation (4) to all features we can obtain a neuron returning equal excitation values for all valid inputs and the minimum excitation for an input vector with some features zeroed. To estimate the maximum excitation if some features are omitted the maximum is subtracted instead of the minimum in equation (4). The resulting transformation of a neuron  $N$  with weights  $W$  and bias  $b$  into neurons  $N_{min}$  and  $N_{max}$  estimating the minimum and maximum excitations, respectively is:

$$\begin{aligned} W'_{min} &= \forall F \in 1 \dots k \quad W'_{minF} = W_F - \min(W_F) \\ b'_{min} &= b - \sum_{F=1}^k \min(W_F). \end{aligned} \quad (7a)$$

$$\begin{aligned} W'_{max} &= \forall F \in 1 \dots k \quad W'_{maxF} = W_F - \max(W_F) \\ b'_{max} &= b - \sum_{F=1}^k \max(W_F). \end{aligned} \quad (7b)$$

Using the neuron transformations (7) it becomes easy to test if the neuron's output is independent of some features. It suffices to zero the inputs associated with the supposedly unimportant features, and check that the maximum and minimum neuron excitations have the same sign.

2) *Multilayer network*: To calculate the minimum excitation of a multilayer network having just one hidden layer we first apply the transformation (7) to all the neurons in the hidden layer. We then start from the output neuron  $ON$  by calculating:

- the minimum excitation of neurons that are connected to  $ON$  with positive weights,
- the maximum excitation of neurons that are connected to  $ON$  with negative weights.

In the case of a more complicated network architecture the above procedure can be repeated recursively. Please note that the transformation (7) gives an exact value for the minimum/maximum excitation of a single neuron. For a multilayer network it gives an upper bound for the minimum excitation and a lower bound for the maximum.

3) *Deriving partial rules from input samples*: The procedure shown in Fig. 2 is used to obtain a partial rule classifying a given sample. The main part of the algorithm searches for a small set of important features. First, an ordering of features has to be selected. Usually, it is the feature ordering of the decision diagram or the ordering determined by feature saliency

```

fun derivePR(net, X)
FeatList ← Ordered features
Important ← Empty
X' ← X
for all f ∈ FeatList do
  X'_f ← 0 {zero the inputs associated with feature}
  if max(net, X') · min(net, X') < 0 then
    {Network's output is unstable}
    add f to Important
    X'_f = X_f {restore the inputs for feature f}
  end if
end for
if Exists F ∈ Important such that class doesn't change
for all values of F then
  remove F from Important
end if
PR ← new partial rule given by equation (2)
return PR

```

Fig. 2. Algorithm to extract a partial rule for a given sample.

for the processed sample.<sup>3</sup> Please refer to the next section, especially II-C1 for suitable algorithms. Then, according to that ordering, the algorithm tries to greedily remove features from the rule's antecedents. It then tries to eliminate one last feature by checking if the network assigns the same class for all its possible values.

### C. Decision diagrams

Up to this point we have treated partial rules as abstract mathematical functions. We now turn our attention to a practical data structure to represent partial rules extracted for each sample and allowing us to efficiently perform the merge operation. The proposed answer comes from the realm of integrated circuits design. A structure called ROBDD (Reduced Ordered Binary Decision Diagram) allowing easy manipulation of boolean functions has been described by Bryant in [12]. Conceptually, a decision diagram is similar to a decision tree: it consists of nodes in which tests for feature values are made and of directed connections between them. An ordering is defined on features and every path in the diagram must traverse the nodes in exactly this order. This facilitates the detection of common subgraphs which can be merged. We will define Reduced Ordered Decision Diagrams (RODDs) as:

**Definition 4.** A Decision Diagram (DD) is a rooted, directed acyclic graph with

- no more than three terminal nodes having out-degree 0 labeled  $\mathcal{T}$ ,  $\mathcal{F}$ , or  $\mathcal{U}$ , and
- a set of feature nodes, in which each feature node  $u$  has an associated feature  $fet(u)$  and the out-degree equal to the number of different values of the associated feature  $n_{fet(u)}$ .

<sup>3</sup>Choosing the global diagram ordering reduces the size of the diagram and has performed better during the experiments. On the other hand, the local feature saliency might result in more general rules.

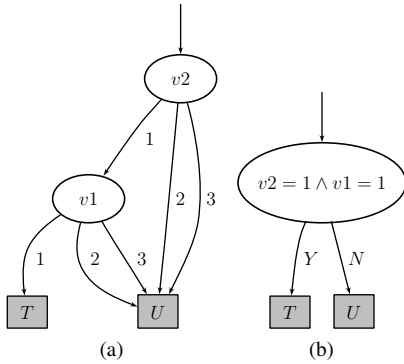


Fig. 3. (a) RODD for function  $\mathcal{T}$  if  $F_1 = 1 \wedge F_2 = 1, \mathcal{U}$  otherwise, (b) same RODD, but pretty-printed.

A DD is Ordered (ODD) if on all paths through the graph the features respect a given linear order  $F_1 < F_2 < \dots < F_k$ . An OBDD is Reduced (RODD) if

- (uniqueness) no two distinct nodes  $u, v$  have the same feature and same successors, i.e.,

$$fet(u) = fet(v), \forall_i succ_i(u) = succ_i(v) \implies u = v,$$

- (non-redundant tests) no feature node  $u$  has all of its successors equal, i.e.,

$$\forall_u \exists_{i,j} succ_i(u) \neq succ_j(u).$$

To implement a function depending on only one feature, only one feature node is needed. Outgoing edges are labeled with possible feature values and point to terminal nodes representing the function values. A diagram implementing a partial rule using the equation (2) has a single path going through all the important features and terminating in the terminal node linked with the desired class. All other edges end in the  $\mathcal{U}$  terminal node. For example, consider a classification problem with 4 features each taking three distinct values 1, 2, 3. Let the feature ordering be  $F_2, F_1, F_3, F_4$ . If a training sample  $S = (1, 1, 2, 3)$  belongs to class  $\mathcal{T}$  and it has been found using the algorithm in Fig. 2 that the important features are  $F_1$  and  $F_2$ , then the partial rule is  $\mathcal{T}$  if  $F_1 = 1 \wedge F_2 = 1$ , otherwise  $\mathcal{U}$  and the resulting diagram is presented in Fig. 3a. A vertical arrow on the top marks the entry-point into the diagram. For the ease of reading bigger diagrams, a graph pretty-printing algorithm has been used and the equivalent diagram is presented in Fig. 3b. Note however, that this only affects presentation, internally the diagram still has two nodes and is not altered in any way.

Fig. 4 shows decision diagrams for the MONK's [13] problems. The diagrams were pretty-printed, merging outgoing edges having the same endpoints and detecting and aggregating conjunctions. Please note the small size of the diagram for the second test – for comparison, an unpruned decision tree can have more than 400 nodes.<sup>4</sup>

The RODDs have many important properties of a good structure to express classification rules derived from neural networks. First, due to node sharing RODDs representing

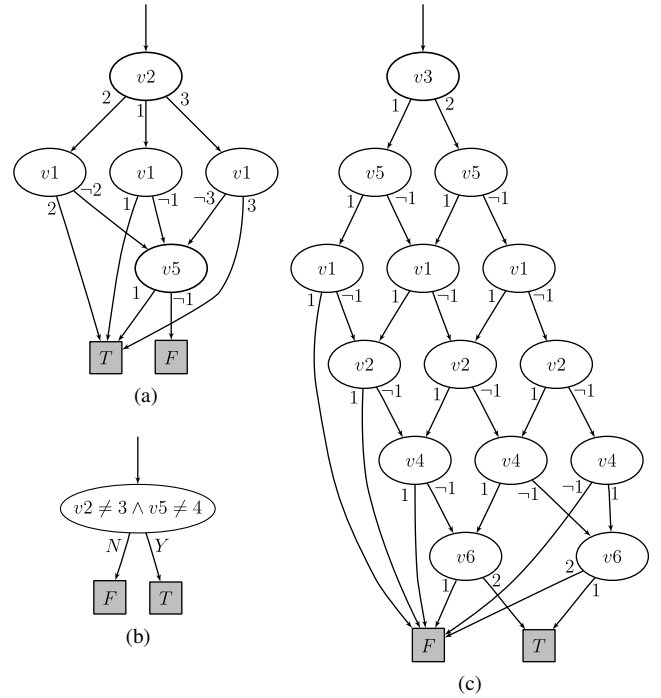


Fig. 4. Decision diagrams for MONK's problems: (a) realizes  $v1 = v2 \vee v5 = 1$ , (b) should realize  $(v5 = 3 \wedge v4 = 1) \vee (v5 \neq 4 \wedge v2 \neq 3)$ , but is instead  $v5 \neq 4 \wedge v2 \neq 3$ . Note that it is internally represented by two feature nodes, which have been merged for better presentation. (c) realizes exactly 2 of  $v1 \dots v6$  are 1.

symmetrical functions (e.g. parity functions, or M-of-N functions) have their size bounded by the square of the number of features. Second, the running time of applying any binary operation (e.g. merge) is bounded by the product of the size of the operands. This means that performing operations on constructed decision diagrams is algorithmically cheap. Popular other data structures for rule handling, such as decision trees or decision tables [14] don't offer such algorithmic complexity guarantees. Also, the diagrams are unique up to feature ordering. This makes it easy to determine whether two DD represent the same function and is the key property in efficient sharing of common expressions.

On the other hand, RODDs have certain drawbacks. First of all, they are highly sensitive to the chosen feature ordering. The same function can require, for different feature orderings, linearly or exponentially many nodes. Also, for some functions (most notably those representing middle bits of integer multiplication) no good ordering exists and the diagrams always require exponentially many nodes. For more information on this interesting data structure, we refer to the tutorial [15].

The use of decision diagrams in machine learning has been studied, sometimes under the name of RODGs (Reduced Ordered Decision Graphs). A method for top-down induction of RODDs is presented by Kohavi in [16]. In [17], Oliveira and Sangiovanni-Vincentelli show an interesting study of a pruning algorithm used for generalization. The suitability of RODDs for visualization of simple decision rules is studied in [18].

We will now describe the main steps necessary to use the decision diagrams structure for rule extraction.

<sup>4</sup>Weka's j48 algorithm generates a tree of size 439, having 298 leaves.

```

fun saliency(net, X)
  Ret[]  $\leftarrow$  zeros(# of features)
  for all F  $\in$  Features do
    for all v  $\in$  values(F) do
      if v = XF then
        skip
      end if
      X'  $\leftarrow$  X; X'F  $\leftarrow$  v
      Ret[F]  $\leftarrow$  |net(X) - net(X')|
    end for
    Ret[F]  $\leftarrow$  Ret[F] / (# of values of F - 1)
  end for
  return Ret[]

```

Fig. 5. The default feature saliency estimation.

```

fun featureOrdering()
  Ordering  $\leftarrow$   $\emptyset$ 
  local fun processCluster(c)
    if c has only one element then
      append c to Ordering
    else
      cs  $\leftarrow$  sub-cluster of c having Strongest(c)
      co  $\leftarrow$  the other sub-cluster of c
      processCluster(cs)
      processCluster(co)
    end if
  end fun
  SumDist[]  $\leftarrow$  0
  SumSaliency[]  $\leftarrow$  0
  for all X  $\in$  Training Set do
    Saliencies[]  $\leftarrow$  saliency(net, X)
    SumSaliency[]  $\leftarrow$  SumSaliency + Saliencies
    SumDist[]  $\leftarrow$  SumDist + distances(Saliencies)
  end for
  Clusters  $\leftarrow$  hierarchicalCluster(SumDist)
  for all c  $\in$  Clusters do
    Strongest[c]  $\leftarrow$  most salient feature in c
  end for
  processCluster(top-level cluster from Clusters)
  return Ordering

```

Fig. 6. Heuristic to determine a variable ordering.

1) *Choosing a feature ordering*: Choosing a good feature ordering is very important. First, the better the ordering, the smaller the resulting diagram will be. Second, the extracted diagram might generalize better, as pruning and generalization procedures work bottom-up and rarely change connections near the top of the diagram. However, choosing an optimal variable ordering is NP-hard and efficient heuristics have to be developed for each application domain.

In the case of a single neuron, a good ordering can be derived from its sorted weights. However, when the network architecture is more complicated it is not obvious how to choose a global ordering. We have developed a heuristic procedure using two assumptions. First, features which often belong to the same rules should be close in the ordering.

TABLE I  
TRUTH TABLE USED FOR THE MERGE OPERATION

<i>A</i>	<i>B</i>	$A \oplus B$
$\mathcal{U}$	<i>X</i>	<i>X</i>
<i>X</i>	$\mathcal{U}$	<i>X</i>
<i>X</i>	<i>X</i>	<i>X</i>
<i>X</i>	<i>Y</i>	$X \neq Y \implies \text{error}$

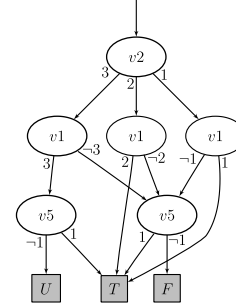


Fig. 7. An incomplete decision diagram for the first MONK's problem after processing a half of the training set.

Second, the most important features should be placed near the top of the diagram.

To determine the saliency of features, the following measures were analyzed: the perturbation method [19], and the maximum, minimum and average difference between the neuron's output for all of a feature's possible values. Using each saliency measure, a partial rule can be generated using the *derivePR(.)* procedure. This rule can be used as a binary measure of feature's importance. The best option proved to be the average difference of network output for all values of a feature (as shown in Fig. 5) and that has been selected as default.

The developed heuristic performs three main steps. First, for every sample belonging to the training set feature saliencies are determined. Distances between these are calculated and added together. Next, a hierarchical clustering algorithm is run on the cumulative distances to find which features are often present together. In each cluster, the most salient feature is determined. Finally, the ordering can be derived. The most salient feature is chosen first. Then, if there are more features in its cluster, the most salient one is selected. Otherwise, the most salient feature from the next cluster is taken. Details are shown in Fig. 6.

2) *Merge operation*: The merge operation has been implemented as a binary operation following the *apply(.)* function pseudo code taken from [15]. The truth table used is shown in Table I. Note that the merge operation should only be run on agreeing diagrams, otherwise the result will be undefined (the merge of disagreeing partial rules results in an error).

3) *Generalization*: If enough training set samples are available and all partial rules have been merged the resulting diagram should cover the whole input space. However, often several nodes point to the  $\mathcal{U}$  terminal node (compare Fig. 4a with Fig. 7), indicating that for some inputs we don't know the answer. There can be several strategies to solve this problem. First, we can construct an example classified by the diagram as  $\mathcal{U}$ , compute its class using the network and add it to

```

fun generalize(Node)
  Chld[] ← children of Node
  FC ← most frequently followed child ≠  $\mathcal{U}$ 
  for all  $c \in \text{Chld}$  do
    if Chld[c] points to  $\mathcal{U}$  then
      Chld[c] ← Chld[FC]
    end if
  end for
  for all  $c \in \text{Chld}$  do {recursively generalize children}
    Chld[c] ← generalize(Chld[c])
  end for
  return Mk(Var(Node), Chld)

```

Fig. 8. Reroute paths leading to  $\mathcal{U}$  to extend the rules domain to whole feature space.

the diagram. This solution stresses network fidelity. However, as stated before, we only want the rules to agree with the network on the training set. A better solution may be to perform some form of heuristic redirecting of paths leading to the  $\mathcal{U}$  node, assuming that if we make the diagram smaller, it will generalize better. A suitable algorithm based on the *simplify*( $\cdot, \cdot$ ) procedure from [15] is presented in Fig. 8. The algorithm assumes that path usage counts have been recorded on the training set prior to its execution. It then proceeds in a top-down manner starting from the root of the diagram. Whenever a child pointer is set to  $\mathcal{U}$ , we change it to point to the most frequently used child. The *mk*( $\cdot$ ) function, described in detail in [15] creates a new node ensuring that the diagram is reduced, i.e. it detects isomorphic subtrees and eliminates nodes whose all children are identical. For example, in Fig. 7, the path  $v2 = 1 \wedge v1 = 1 \wedge v5 \neq 1 \rightarrow \mathcal{U}$  would be changed to  $v2 = 1 \wedge v1 = 1 \wedge v5 = \text{any} \rightarrow \mathcal{T}$ , which in turn would be simplified by *mk*( $\cdot$ ) to  $v2 = 1 \wedge v1 = 1 \rightarrow \mathcal{T}$ . In this case the heuristic would make the right choice.

4) *Pruning*: In many cases, the generated rules do correctly classify all samples, but they are overly complex. The pruning procedure is used to reduce the size of the diagram, while preserving its output on all training samples, or a majority of them. The main idea is simple: first, count how often a path was selected for all the elements in training set. Second, change the seldom (according to a selected threshold) or never used child pointers to point to the  $\mathcal{U}$  node. Third, run the generalization procedure. In the current implementation, pruning is repeated as long as it reduces the size of the diagram.

5) *Various enhancements*: Several simple heuristics have been applied during the development of the software to improve its performance. The most important one controls how much information is extracted from a single sample. First, instead of trying to eradicate only one last feature by checking all its possible values in the *generalize*( $\cdot$ ) procedure (Fig. 2), we can mark for each important feature all its values that don't change the network output and merge them into the diagram. Second, we can check the whole sample's neighborhood (in the Hamming distance sense) and if some parts of it do not belong to the domain of the current rule, we can derive a new partial rule. Optionally, we can only add rules from

```

fun printDNF(DD)
if articulation point  $P$  present in  $DD$  then
  divide  $DD$  on node  $P$ 
  printDNF(nodes above  $P$ )
  printDNF(nodes below  $P$ )
else
   $R \leftarrow \text{shortestPathToT}(DD)$ 
  print  $R$  as a clause
   $D' = \text{simplify}(\text{not}R, DD)$ 
  printDNF( $D'$ )
end if

```

Fig. 9. Algorithm for printing rules as a DNF formula from a decision diagram.

neighboring samples belonging to a different class.

6) *DNF rule extraction from a diagram*: While the RODDs are graphs easy to read and interpret, a user might be interested in Disjunctive Normal Form (DNF) logical rules. Direct listing of all paths leading to the desired terminal node produces sub-optimal results. For instance, for the first MONK's problem, if the extracted diagram matched the one presented in Fig. 4a, the simplest rule  $v5 = 1$  would be cluttered with the expansion of  $v1 \neq v2$ . To work around this problem articulation points<sup>5</sup> are found in the diagram. In the example, the node for the feature  $v5$  is an articulation point for the paths to  $\mathcal{F}$  (this is indicated by the bold oval around the node). Such an articulation point shows a logical *or* composition: if the output is  $\mathcal{F}$ , both the first part (above the articulation point) and the second part (below) must fail. We can thus divide the diagram and extract the rules separately. In this way, we get 4 rules:  $v5 = 1 \vee v1 = 1 \wedge v2 = 1 \vee v1 = 2 \wedge v2 = 2 \vee v1 = 3 \wedge v2 = 3$ . Details are shown in Fig. 9. The *simplify*( $\cdot, \cdot$ ) function is described in [15].

The graph structure makes it possible to implement more complicated mechanisms, such as detecting equality or M-of-N conditions. For example, when the diagram is printed, we try to detect chains of consecutive tests leading to the same node and present them in an aggregate form, as in Fig. 4b.

### III. TEST RESULTS

The LORE algorithm was first tested on 100 randomly generated, simple logical formulas. Each formula had 6 features and was expressed in a DNF form having 8 clauses, each containing on the average 3.5 literals. For each formula, the best ordering had been found by testing all the possibilities and the smallest diagram size was compared to the one obtained with the heuristic for feature ordering. Results have been shown in Table II. On the average, the feature ordering heuristic produced graphs having 9.28 nodes, while the average size of the diagrams for a random ordering is 10.58 with a standard deviation 1.45. Hence the feature ordering selected by the heuristics produced on the average diagrams one standard deviation smaller than those using a random ordering. For 58 out of 100 cases, the heuristic procedure

<sup>5</sup>These are nodes through which every path to a certain terminal node must pass.

TABLE II  
EFFICIENCY OF THE FEATURE ORDERING HEURISTIC.  
AVERAGED RESULTS FOR 100 RUNS.

Min size	Max size	Avg. size	Standard deviation	Heuristic size
8.47	14.58	10.89	1.45	9.28

TABLE III  
RESULTS ON THE MONK'S TESTS.

Test ID	Network			LORE rules unpruned			LORE rules pruned		
	errors	time [s]	HL size	errors	size	time [s]	errors	size	time [s]
1	0	3	4	0	7	0.4	0	7	0.6
2	0	3.2	4	7.1%	39	0.44	18.5%	19	0.93
2 <sup>a</sup>	0	3.2	4	0	16	0.45	0	16	0.67
3	4.6%	1.4	1	4.6%	9	0.5	2.8%	4	0.9
3 <sup>b</sup>	2.8%	0.74	1	2.8%	4	0.38	2.8%	4	0.6

<sup>a</sup>Neighboring samples of opposite class were analysed.

<sup>b</sup>Network was regularized using a high weight decay coefficient.

TABLE IV  
RESULTS ON THE VARIOUS TESTS ON DATA FROM THE UCI REPOSITORY.

Test name	Network			LORE rules unpruned			LORE rules pruned			J48 unpruned		J48 pruned		avg. time [s]
	errors	HL size	time [s]	errors	size	time [s]	errors	size	time [s]	errors	size	errors	size	
mushrooms <sup>a</sup>	0	10	17.32	0	114	20.47	0	11.32	20.81	0	28.24	0	28.24	1.23
voting <sup>a</sup>	13.4	10	1.26	12.8	36.28	0.38	12.6	19.8	0.68	14.2	23.24	32.8	5.08	0.04
krkpa7 <sup>b</sup>	21	35	128	138	4597	11.69	153.6	605	12.04	15.4	88.6	75	52.12	0.48
promoter <sup>c</sup>	9.2	1	1.15	15	28.84	0.53	17.2	15.72	0.87	26.4	37.96	49.8	19.24	0.02
promoter <sup>cd</sup>	7.2	1	1.23	8.2	331	2.1	18.2	15.08	2.5	26.6	36.84	41.6	21	0.02
promoter <sup>ce</sup>	6	1	1.32	9	26.18	0.6	11	16.31	0.92	24	44.51	442	24.74	0.02
promoter <sup>cde</sup>	6	1	1.35	2	388.99	2.69	14	16.83	3.15	24	44.51	442	24.74	0.02

<sup>a</sup>Network trained using standard backpropagation.

<sup>b</sup>Network trained with weight decay, ratio=0.9

<sup>c</sup>Network trained with weight decay, ratio=0.3

<sup>d</sup>Merging in rules from training set samples neighborhood

<sup>e</sup>Tested using the leave-one-out methodology

resulted in diagrams having the minimum size. Furthermore, it never led to a diagram having the maximum possible size.

As a second test, the ubiquitous MONK's [13] problems have been used. The three tests use the same data, consisting of six categorical features,  $v_1 \dots v_6$ , and only the relation used to classify samples is changed. In the first test it is  $v_1 = v_2 \vee v_5 = 1$ , in the second it is *exactly 2 features are 1*, and in the third  $(v_5 = 3 \wedge v_4 = 1) \vee (v_5 \neq 4 \wedge v_2 \neq 3)$ . Moreover, in the third test five percent of training samples have incorrect labels. In all the tests neural networks were trained using weight decay. Also, the diagram was pruned by removing paths used less than 3 times. The results are shown in the Table III. Good decision diagrams are shown in Fig. 4. We can see that while there is enough training set samples to generate a diagram to cover whole input space in the first and third tests, in the second test the diagram is incomplete – even though the network has learned without errors, the decision diagram has a high error-rate. In this case, adding information about the neighborhood of a sample solved the issue. Compare Fig. 4c showing a good diagram with Fig. 10a showing a diagram before generalization and Fig. 10b depicting the results of pruning. In the third test, the training set is intentionally corrupted with noise. We can observe that prior to pruning, the diagram exactly mimics the network (the error rate is the same). The learned relation is more complicated than the expression used to generate the data. Retraining the network with a more aggressive weight decay or pruning the diagram both lead to a diagram which generalizes better, but only one clause out of two is properly detected. Compare Fig. 10c showing an unpruned diagram with the pruned one in Fig. 4b.

The next four tests used data from the UCI repository [20]: the mushroom data set, the congressional voting records data set, the chess king-rook vs. king-pawn data set (further named krkpa7) and the molecular biology promoter gene sequence data set. Since LORE doesn't support missing values, samples from the voting records data set containing missing values were rejected. In the mushroom set, the missing values were left as a new feature value "?". For all the sets, unless noted otherwise, five runs of full five-fold cross validation were executed and the results have been averaged. Also, pruning has been set up to preserve 100% accuracy on the training set. To compare the effectiveness of rule extraction, the C4.5 algorithm was run using Weka's J48 implementation [21]. Its parameters were *-U -M1* for runs without pruning and *-C0.25 -M2* for runs with pruning. The results have been presented in Table IV. Diagram and tree sizes were used to compare the understandability of extracted rules.

The mushroom and voting sets were used to demonstrate the "out-of-the-box" capabilities of LORE. The neural network was trained using reasonable parameters chosen based on our experience. No tuning of network parameters was performed to demonstrate how LORE performs on imperfect networks trained in a most typical way. In both cases pruning results in better rules – the diagrams are smaller and retain comparable accuracy to the pruned ones. Also, the pruned diagrams perform better than both the network and the decision trees, while being smaller than the decision trees. It is worth noting that in both cases LORE running times are comparable to that of network training. This coincides with our estimation of the computational complexity of the method given in Appendix B. The decision tree, however, is induced an order of magnitude

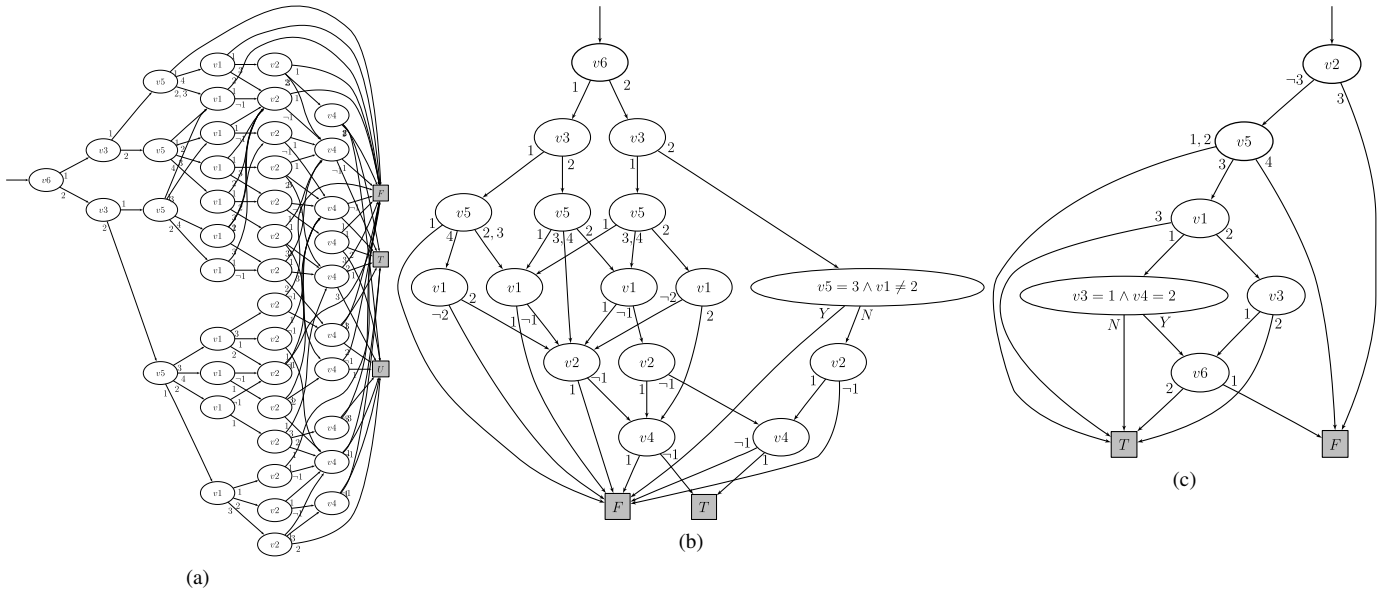
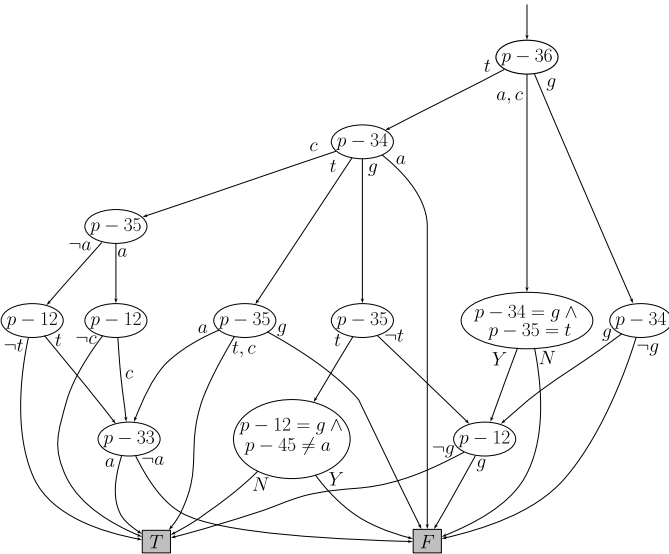


Fig. 10. Erroneous decision diagrams for the Monk's problems: (a) test 2 before diagram generalization, (b) test 2 after pruning, and (c) test 3 prior to pruning.



Extracted rules for class  $\mathcal{T}$ :

- $p - 36 = t \wedge p - 35 = t, c \wedge p - 34 = t$
- $p - 36 = g \wedge p - 34 = g \wedge p - 12 \neq g$
- $p - 36 = t \wedge p - 35 = a \wedge p - 34 = t \wedge p - 33 = a$
- $p - 36 = t \wedge p - 35 = t \wedge p - 34 = g \wedge p - 12 \neq g$
- $p - 45 = a \wedge p - 36 = t \wedge p - 35 = t \wedge p - 34 = g$
- $p - 36 = t \wedge p - 35 \neq t \wedge p - 34 = g \wedge p - 12 \neq g$
- $p - 36 = t \wedge p - 35 \neq a \wedge p - 34 = c \wedge p - 12 \neq t$
- $p - 36 = t \wedge p - 35 \neq a \wedge p - 34 = c \wedge p - 33 = a$
- $p - 36 = t \wedge p - 35 = a \wedge p - 34 = c \wedge p - 12 \neq c$
- $p - 36 = t \wedge p - 35 = a \wedge p - 34 = c \wedge p - 33 = a$
- $p - 36 = a, c \wedge p - 35 = t \wedge p - 34 = g \wedge p - 12 \neq g$

Fig. 11. A typical diagram and the rules in DNF form extracted for the promoter domain problem without the use of neighboring samples and with pruning enabled.

faster than network training and rule extraction.

The results on the krkpa7 set are disappointing. The generated diagrams are not only big and hard to understand, but also their performance is worse than that of the network or of the decision trees. However, the issues may stem from some incompatibility between this data set and neural networks. The data set requires a network of huge size (networks with few hidden neurons do not learn the relation well), with long training times (mainly due to the weight decay mechanism used), only to achieve a performance worse than that of an unpruned decision tree. We include the results from this data set, however, to show that the method execution time is acceptable even for larger networks.

The promoter data set was first introduced to test the effectiveness of the KBANN method. Results published in [8]

for a leave-one-test state that the ID3 method makes 19 errors in 106 runs, neural networks under standard backpropagation make 8/106 and the KBANN method achieves the lowest rate of 4/106 errors. For the purpose of this comparison, we tested our method under the same conditions. However, since a single neuron with weight decay performs better than the described network, we have chosen it as the base for rule extraction. We can observe that the proposed method under the default settings extracts understandable rules and has a slightly worse accuracy than the network (prior to pruning). However, when the information from neighboring samples is included, the rules before pruning show a low error rate of 2 in 106 runs, at the cost of rule legibility. This demonstrates that the idea of using a network to discover rules in close proximity to training set samples is useful, however better

pruning and diagram generalization algorithms are needed. A sample diagram and accompanying rules extracted during the leave-one-test without the use of neighboring samples and after pruning is shown in Fig. 11.

#### IV. CONCLUSIONS

The LORE method of rule extraction from neural networks uses many novel ideas. First, the proposed approach focuses on retaining high network fidelity on the training set, while allowing the rule set to diverge from the network in the remaining feature space, making it possible to reconcile the dilemma whether one seeks good network fidelity or accuracy.

Another achievement is the adaptation of the reduced, ordered decision diagram data structure to support the merge and generalization algorithms. This, in its own merit, might prove to be a valuable tool in other rule induction schemes.

We have shown a mathematically sound method of presenting the domain of a generated rule set. Adaptations of this technique to other rule extraction methods might help to distinguish between errors in the rules (which result in false positive classifications) and incompleteness of the rules (which result in false negative classifications). In our method, this technique was a key step in the design of algorithms that simplify the rule set without loss of training accuracy.

Future work will first be directed towards the development of better pruning algorithms. The minimum description length principle (already investigated in the context of decision diagrams in [17]) might become a valuable tool for increasing the generalization abilities of decision diagrams, while at the same time reducing their size.

Improved rule presentation is another important topic worth continued studying. The presented transformation of RODDs into DNF format rules shows that the RODDs might be used as intermediate representation purely for their algorithmic properties and the final rules presented to the user may be expressed in a more understandable form of decision trees, decision tables or rules.

#### APPENDIX A

##### COMPUTATIONAL COMPLEXITY OF RULE EXTRACTION FROM NEURAL NETWORKS

Before we analyze the complexity of rule extraction methods, we will try to capture the meaning of what an “understandable” rule set is. We propose the following definition:<sup>6</sup>

**Definition 5.** *A rule set is usable if we can classify in polynomial time a previously unknown sample. Moreover, it is understandable if, for a given class, we can show in polynomial time a sample input belonging to that class. If, in polynomial time, we can show the smallest set of features a sample must have to belong to a class, we will say that the rule set is very understandable.*

The *usable* rule sets are all those that can be practically used to classify new samples. Under this definition, the network

<sup>6</sup>All the reasoning in this section assumes that  $P \neq NP$  and hence problems that we can not solve in polynomial time are unsolvable.

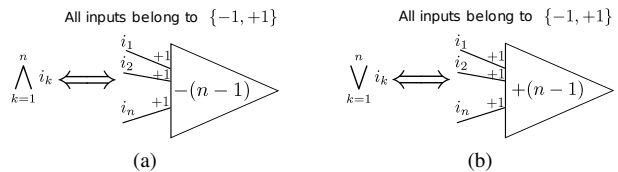


Fig. 12. Implementing the boolean functions (a) *and* and (b) *or* with a neuron.

itself is usable. We believe that this the minimum requirement extracted rules must meet to be of any practical use.

The *understandable* rule sets are those, for which we can show examples belonging to a particular class. Decision trees surely meet this criterion – we just have to trace a path from an interesting leaf node to the top of the tree. However, the DNF formulas are not understandable, because showing examples for the 0 (false) class is equivalent to showing the satisfiability of a CNF formula (as the negation of DNF is CNF), which is a NP-hard problem. Similarly, if one doesn’t merge partial rules extracted for every unit inside a network, showing an example for any class is NP-hard in general.

One can argue that the ability to extract examples from the rules is not important, because the data set contains many of them. The *very understandable* class of rules tries to capture the ability of showing the simplest example for a class (or equivalently the shortest clause if we were to write down the rules in the DNF form). This definition was chosen to resemble some of the actions a person trying to understand an unknown object would do, i.e. see how it reacts to a given input, find other inputs causing the same reaction and finally, try to find the simplest common factor such inputs may have.

The next two theorems show that if we can only use a given network (either pedagogically by looking only at its output, or deductively, by also looking at its structure), then finding understandable and very understandable rules perfectly mimicking the network is an NP-hard problem.

**Theorem 2.** *Extraction of an understandable rule set exactly describing a given neural network is NP-hard.*

*Proof:* We will reduce the NP-complete satisfiability of CNF formulas to the rule problem of extraction from a neural network. The variables of the formula become the inputs. Every hidden layer’s neuron implements the logical *or* function and represents a single clause. The output neuron implements the logical *and* function to provide the conjunction of clauses. In Fig. 12 it is shown how to express both functions as a neuron.

The described reduction has a polynomial time complexity. If we can also use the extracted rules to find in polynomial time an example for which the class is “true”, then the rule extraction has to be harder than the satisfiability problem. ■

**Theorem 3.** *It is NP-hard to find a very understandable rule set exactly describing a given network, even if the network’s function satisfiability is assumed.*

*Proof:* We will find a reduction for the dominating-set problem. As an example, in Fig. 13 a neural network

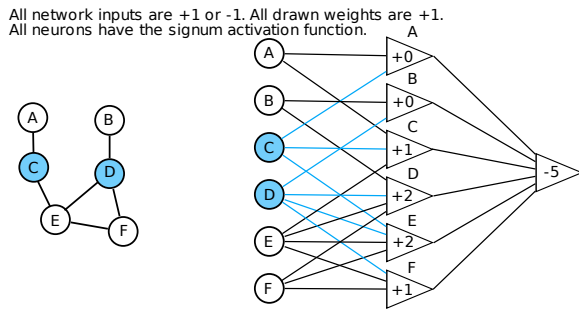


Fig. 13. Solving the NP-hard Dominating Set problem can be reduced to finding the smallest set of inputs causing the output neuron to fire.

corresponding to a small graph is shown. For every node of the given graph, we assign a boolean input feature and a hidden layer neuron. An input is connected to a hidden layer neuron if and only if there is an edge between the vertices represented by this input and hidden layer neuron or they represent the same vertex. Thus every hidden layer neuron implements an *or* function. We then set the output neuron to implement an *and* function.

Clearly, the function is satisfiable, as an input of all ones (meaning that we select all the vertices as the dominating set) causes the output neuron to fire. However, finding the smallest set of inputs causing the output neuron to fire is equivalent to selecting the smallest dominating set of graph nodes. ■

It should be noted that the first problem is simply solved by looking at the training set. Intuitively, if a network has learned a complex relation as in the proof of the second theorem, there must be some training samples close to the decision boundary, and using them for rule extraction is just what we need.

## APPENDIX B COMPUTATIONAL COMPLEXITY OF LORE

The most costly operation performed is the merge of partial rules. Its running time is proportional to the product of the size of the diagrams being merged. The pessimistic size of resulting RODDs is similar to the case of representing simple monotone 2CNF formulas (i.e. having clauses with only two literals and no negations), which have exponentially many nodes [22]. Hence in the worst case the algorithm performs exponentially many operations.

### AVAILABILITY OF SOFTWARE

Currently a mixed Matlab-Java implementation is available from authors upon request. It requires Matlab R2008a, Java 5, and the Graphviz Dot utility. If sufficient need arises, we are willing to release the software as a Weka plug-in.

### ACKNOWLEDGMENT

The authors thank anonymous reviewers who have helped improve the presentation of this article and offered many useful and insightful suggestions. They also thank Michał Gorzelany, Artur Abdullin, and Jordan Malof for useful discussions.

## REFERENCES

- [1] R. Andrews, J. Diederich, and A. B. Tickle, "Survey and critique of techniques for extracting rules from trained artificial neural networks," *Knowledge-Based Systems*, vol. 8, no. 6, pp. 373–389, 1995.
- [2] A. B. Tickle, R. Andrews, M. Golea, and J. Diederich, "The truth is in there: directions and challenges in extracting rules from trained artificial neural networks," *IEEE Transactions on Neural Networks*, vol. 9, pp. 1058–1068, 1998.
- [3] M. W. Craven, "Extracting comprehensible models from trained neural networks," Ph.D. dissertation, University of Wisconsin–Madison, 1996.
- [4] T. Etchells and P. Lisboa, "Orthogonal search-based rule extraction (osre) for trained neural networks: a practical and efficient approach," *Neural Networks, IEEE Transactions on*, vol. 17, no. 2, pp. 374–384, march 2006.
- [5] L. M. Fu, "Rule generation from neural networks," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 24, no. 8, pp. 1114–1124, 1994.
- [6] R. Krishnan, G. Sivakumar, and P. Bhattacharya, "A search technique for rule extraction from trained neural networks," *Pattern Recognition Letters*, vol. 20, no. 3, pp. 273–280, 1999.
- [7] R. Setiono, B. Baesens, and C. Mues, "Recursive neural network rule extraction for data with mixed attributes," *Neural Networks, IEEE Transactions on*, vol. 19, no. 2, pp. 299–307, feb. 2008.
- [8] G. Towell, J. Shavlik, and M. Noordewier, "Refinement of approximate domain theories by knowledge-based neural networks," in *Proceedings of the Eighth National Conference on Artificial Intelligence*. Citeseer, 1990, pp. 861–866.
- [9] D. W. Opitz and J. W. Shavlik, "Dynamically adding symbolically meaningful nodes to knowledge-based neural networks," *Knowledge-Based Systems*, vol. 8, no. 6, pp. 301–311, 1995.
- [10] N. Barakat and J. Diederich, "Eclectic rule-extraction from support vector machines," *International Journal of Computational Intelligence*, vol. 2, no. 1, pp. 59–62, 2005.
- [11] Z. H. Zhou, "Rule extraction: Using neural networks or for neural networks?" *Journal of Computer Science and Technology*, vol. 19, no. 2, pp. 249–253, 2004.
- [12] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 35, pp. 677–691, 1986.
- [13] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. D. Jong, S. Dzeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. V. D. Welde, W. Wenzel, J. Wnek, and J. Zhang, "The monk's problems a performance comparison of different learning algorithms," Tech. Rep., 1991.
- [14] B. Baesens, R. Setiono, C. Mues, and J. Vanthienen, "Using neural network rule extraction and decision tables for credit-risk evaluation," *Management Science*, pp. 312–329, 2003.
- [15] H. R. Andersen, "An introduction to binary decision diagrams," IT University of Copenhagen, Lect. Not., 1999. [Online]. Available: <http://www.itu.dk/people/hra/notes-index.html>
- [16] R. Kohavi and C. H. Li, "Oblivious decision trees graphs and top down pruning," in *Proceedings of the 14th international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc., 1995, pp. 1071–1077.
- [17] A. L. Oliveira and A. Sangiovanni-Vincentelli, "Using the minimum description length principle to infer reduced ordered decision graphs," *Machine Learning*, vol. 25, no. 1, pp. 23–50, 1996.
- [18] C. Mues, B. Baesens, C. M. Files, and J. Vanthienen, "Decision diagrams in machine learning: an empirical study on real-life credit-risk data," *Expert Systems with Applications*, vol. 27, no. 2, pp. 257–264, 2004.
- [19] J. M. Zurada, A. Malinowski, and S. Usui, "Perturbation method for deleting redundant inputs of perceptron networks," *Neurocomputing*, vol. 14, no. 2, pp. 177–193, 1997.
- [20] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [21] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The WEKA data mining software: An update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.
- [22] M. Langberg, A. Pnueli, and Y. Rodeh, "The robdd size of simple cnf formulas," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2860, pp. 363–377, 10.1007/978-3-540-39724-3\_32.



**Jan Chorowski** received the M.Sc degree in electrical engineering from the Wrocław University of Technology, Poland. He is the recipient of the University Scholarship and is currently working towards his Ph.D. at the University of Louisville, Kentucky. His research interests include the development of machine learning algorithms, especially using neural networks.



**Jacek M. Zurada** received the MS and PhD degrees (with distinction) in electrical engineering from the Technical University of Gdansk, Poland, in 1968 and 1975, respectively. Since 1989, he has been a professor in the Department of Electrical and Computer Engineering, University of Louisville, Kentucky. He was the department chair from 2004 to 2006. He was an associate editor of the IEEE Transactions on Circuits and Systems, Part I and Part II, and served on the editorial board of the Proceedings of IEEE. From 1998 to 2003, he was the editor-in-chief of the IEEE Transactions on Neural Networks. He is an associate editor of Neural Networks, Neurocomputing, Schedae Informaticae, and the International Journal of Applied Mathematics and Computer Science, the advisory editor of the International Journal of Information Technology and Intelligent Computing, and the editor of Springer's Natural Computing Book Series. He has served the profession and the IEEE in various elected capacities, including as the President of the IEEE Computational Intelligence Society in 2004-2005. He now chairs the IEEE TAB Periodicals Committee (2010-11) and the IEEE TAB Periodicals Review Committee (2012-13). He is a Distinguished Speaker of IEEE CIS and a Fellow of the IEEE.