# EXTRACTING BOOLEAN AND PROBABILISTIC RULES FROM TRAINED NEURAL NETWORKS

**Pengyu Liu**
Bioinformatics Center, Institute for Chemical Research
Kyoto University
Gokasho, Uji, Kyoto, Japan 611-0011
liupengyu@kuicr.kyoto-u.ac.jp

**Avraham A. Melkman**
Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel 84105
melkman@cs.bgu.ac.il

**Tatsuya Akutsu**
Bioinformatics Center, Institute for Chemical Research
Kyoto University
Gokasho, Uji, Kyoto, Japan 611-0011
takutsu@kuicr.kyoto-u.ac.jp

December 3, 2020

**ABSTRACT**

This paper presents two approaches to extracting rules from a trained neural network consisting of linear threshold functions. The first one leads to an algorithm that extracts rules in the form of Boolean functions. Compared with an existing one, this algorithm outputs much more concise rules if the threshold functions correspond to 1-decision lists, majority functions, or certain combinations of these. The second one extracts probabilistic rules representing relations between some of the input variables and the output using a dynamic programming algorithm. The algorithm runs in pseudo-polynomial time if each hidden layer has a constant number of neurons. We demonstrate the effectiveness of these two approaches by computational experiments.

***Keywords*** neural networks boolean functions rule extraction dynamic programming

## 1 Introduction

Recent developments in deep learning technologies have demonstrated the power of artificial neural networks in making predictions in various areas. It is therefore of great interest to develop a methodology for interpreting how a trained neural network arrives at its predictions: better understanding makes for greater trust. There are two major approaches to the interpretation of trained networks: mechanism explanation and behavior explanation. The former sees the network as a white box and aims to explain its inner workings. The latter settles for explaining what outputs will be obtained from the different inputs, viewing the network as if it were a black box. Many studies adopt the latter approach because it is easier and useful.

Different methodologies can be broadly divided as follows:

- Boolean function extraction: Extract a Boolean function from a trained neural network (Tsukimoto, 2000).
- Activation maximization: Create an input to maximize the network output and show a representative example of a specific category (Le, 2013).
- Sensitivity analysis: Examine the sensitivity of the network against changes of each input (Smilkov et al., 2017).
- Deconvolution: Trace the path from the output to the input and analyze attributes learned by a convolutional neural network (Springenberg et al., 2014).

- LIME: Explain the predictions by approximation with an interpretable model (Ribeiro et al., 2016).

We focus here on Boolean function extraction. Most such extraction algorithms fall into of two categories: decompositional or pedagogical algorithms.

Decompositional algorithms such as (Tsukimoto, 2000) extract a Boolean function for each individual neuron, and merge these rules into a set of rules describing the behavior of the neural network . Such algorithms generate their results in terms of the structure of the network so that we can understand how each neuron works.

Pedagogical algorithms, such as (Augusta and Kathirvalavakumar, 2012) and (Saad and Wunsch II, 2007), view rule extraction as the task of learning a Boolean function from data samples. Although these methods generate accurate results, the results may not reflect the function of each neuron.

This paper is motivated by Tsukimoto's work (Tsukimoto, 2000). His algorithm extracts Boolean functions in disjunctive normal form from each of the neurons independently and then combines these functions. It works in polynomial time if the size of each term is bounded by a constant. In this paper we explore several extensions and modifications of this pioneering work, motivated by the following considerations.

First, Tsukimoto's algorithm requires the specification of the maximum size of a term; if that size is too small the algorithm may fail to extract a Boolean function, and if it is large, the running time may be too long. Second, for a neural network with hidden layers the size of the Boolean function may be large.

To begin addressing these issues we propose two algorithms; admittedly, further refinements will be needed to make them practical for large-scale neural networks. The first one extracts Boolean functions in the form of nested canalyzing functions (equivalent to 1-decision lists), majority functions, or certain combinations of the two. This algorithm is useful if each neuron indeed has such a form. Otherwise, as for Tsukimoto's algorithm, the algorithm may output large Boolean functions. This phenomenon is inevitable if exact prediction rules are to be extracted in the form of Boolean functions. The second algorithm extracts rules in the form of conditional probabilities. Each conditional probability represents a relation between an input attribute, or a combination of input attributes, and an output. Although the prediction rules obtained by this approach are usually not exact they can be concise. The potential usefulness of these two algorithms are demonstrated by means of computational experiments.

## 2 Preliminaries

### 2.1 Neural Networks

In this paper a neuron is a linear threshold function (O'Donnell, 2014), with the exception of Section 5.2, in which we consider also neurons that are sigmoid functions.

**Definition 1.** *Given $\theta, w_1, \cdots, w_n \in \mathbb{R}$, a linear threshold function is a Boolean-valued function $f : \{0,1\}^n \to \{0,1\}$ defined by*

$$f(x) = H(w_1x_1 + \cdots + w_nx_n - \theta)$$

*where $H(z)$ is the Heaviside function*

$$H(z) = \begin{cases} 1 & z \geq 0, \\ 0 & z < 0. \end{cases}$$

We will assume that all weights and threshold values are integers. This is a reasonable assumption in practice inasmuch as a computer handles only rational numbers, and a threshold function with rational weights can be transformed into an equivalent threshold function with integer weights.

A sigmoid function is in general characterised by its "S"-shaped curve, but here we limit it to the logistic function:

**Definition 2.** *The one-dimensional sigmoid function is*

$$S(x) = \frac{1}{1 + e^{-x}}.$$

*More generally, given $\theta, w_1, \cdots, w_n \in \mathbb{R}$, the $n$-dimensional sigmoid function is $S(w_1x_1 + \cdots + w_nx_n - \theta)$.*

**Definition 3.** *In this paper, a neural network is a directed* acyclic *graph with neurons situated at the vertices. The neurons with zero indegree are input neurons, which receive signals only from the outside world, and those with zero outdegree are output neurons.*

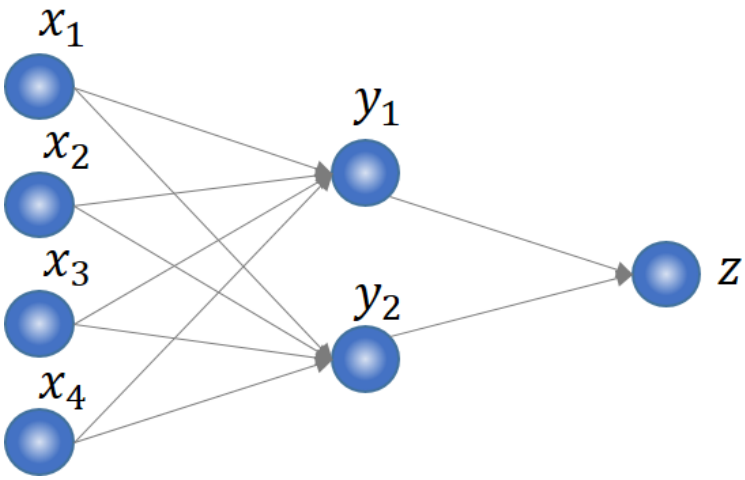


Figure 1: Example of a neural network.

Fig. 1 shows an example of a neural network in which $x_1, x_2, x_3, x_4$ are the input neurons and $z$ is the output neuron. Since the input values are 0 or 1 the output value is determined by a Boolean function of the input values. The problem we address is the extraction of this Boolean function in a simple form from a given neural network.

### 2.2 Majority and Nested Canalyzing Functions

For Boolean variables $x$ and $y$, $\bar{x}$, $x \wedge y$, $x \vee y$ denote the negation of $x$, conjunction of $x$ and $y$, and disjunction of $x$ and $y$, respectively. A literal is either a Boolean variable or its negation. A term is a conjunction of literals, and a conjunction of $k$ literals is called a $k$-term.

**Definition 4.** *Given $n$ literals $\ell_1, \ldots, \ell_n$ and $k \in \{1, \ldots, n\}$, the $k$-out-of-$n$ majority function $Maj_k(\ell_1, \ldots, \ell_n)$ is the Boolean function that is the disjunction of all conjunctions of exactly $k$ out of the $n$ literals. Thus its output is 1 when $k$ or more literals are 1, and 0 otherwise.*

**Definition 5.** *A Boolean function $f(x)$ is nested canalyzing if it can be represented as*

$$f(x) = \ell_1 \vee \cdots \vee \ell_{k_1-1} \vee (\ell_{k_1} \wedge \cdots \wedge \ell_{k_2-1} \wedge (\ell_{k_2} \vee \cdots \vee \ell_{k_3-1} \vee (\cdots)))$$

*where $\ell_i \in \{x_1, \bar{x}_1, \cdots, x_n, \bar{x}_n\}$ and $1 \leq k_1 < k_2 < \cdots$. It can be assumed that each $x_i$ appears at most once in $f(x)$ either positively or negatively.*

It has been reported that many biologically relevant functions are nested canalyzing ones (Harris et al., 2002; Jarrah et al., 2007). Therefore, extracting rules in the form of nested canalyzing functions is meaningful. Next we define the decision list (Anthony, 2010):

**Definition 6.** *Let $M$ be a set of Boolean functions on $\{0,1\}^n$. The decision list $f$ on $M$ is a finite sequence:*

$$f = \langle (f_1, c_1), (f_2, c_2), \cdots, (f_r, c_r) \rangle,$$

*such that $f_i \in M$ and $c_i \in \{0,1\}$ for $i = 1, \ldots, r$. The value of $f$ is defined by $f(y) = c_j$ where $j = \min\{i | f_i(y) = 1\}$, or 0 if there is no $i$ such that $f_i(y) = 1$.*

*A 1-decision list is a decision list in which each $f_i$ is a literal.*

The decision list $f$ is like a sequence of "if then else" commands:

**if** $f_1(y) = 1$ **then** $f(y) = c_1$
**else if** $f_2(y) = 1$ **then** $f(y) = c_2$
.
.
**else if** $f_r(y) = 1$ **then** $f(y) = c_r$
**end if**

**Proposition 7** (Theorem 2.8 of (Jarrah et al., 2007).)**.** *The class of 1-decision lists is equivalent to the class of nested canalyzing functions.*

**Proposition 8.** *The number of nested canalyzing functions on $n$ variables is bounded above by $n! \cdot 5^n$.*

*Proof.* There are $n!$ possible orderings of variables. Consider a particular order, say $x_1, x_2, x_3, \cdots, x_n$. The number of functions with this order, can be bounded recursively from the following five possibilities for $f(x_1, \cdots, x_n)$:

- $x_1 \wedge g(x_2, \cdots, x_n)$,
- $\bar{x}_1 \wedge g(x_2, \cdots, x_n)$,
- $x_1 \vee g(x_2, \cdots, x_n)$,
- $\bar{x}_1 \vee g(x_2, \cdots, x_n)$,
- $x_1$ is not used.

Therefore, there are at most $n! \cdot 5^n$ possibilities in total. □

A more precise but more complicated bound is given in (Jarrah et al., 2007). Since every nested canalyzing function can be represented as a linear threshold function (Anthony, 2001), and the number of linear threshold functions on the Boolean domain is at least $2^{n(n-1)/2}$ (Anthony, 2001), what these bounds show is that the class of nested canalyzing functions is a proper subset of the class of linear threshold functions.

## 3 Extraction of Boolean Functions

### 3.1 Extraction of a Nested Canalyzing Function or a Majority Function

In this subsection, we consider first the extraction of a rule having the form of a nested canalyzing function, and then the extraction of a rule in the form of a nested majority function.

When analyzing the running times of algorithms we will use $O^*(f(N))$ to represent $O(f(N)poly(N))$, where $N$ is the total size of input data. We assume that the size of each parameter is polynomially bounded with respect to $n$, where $n$ denotes the number of input variables. Recall that addition, subtraction, and multiplication can be done in time polynomial in the total size of the terms involved.

But first let us present a simple but very useful Lemma.

**Lemma 9.** *For the purpose of rule extraction from a linear threshold function it can be assumed without loss of generality that the threshold function has the form $H(w_1x_1 + \cdots + w_nx_n - \theta)$ with $w_1 \geq \cdots \geq w_n > 0$.*

*Proof.* Repeat the following step until no negative weights remain: if $w_i < 0$ for some $i$, replace $\theta$ by $\theta - w_i$, $w_i$ by $-w_i$, and $x_i$ by $\bar{x}_i$. These two threshold functions are clearly equivalent.

Next, reorder the resulting linear threshold function so that the terms $w_ix_i$ appear in descending order of $w_i$.

Finally, for convenience, replace each $\overline{x_i}$ by $x_i$. After extraction of the rule this replacement can be reversed. □

To illustrate the procedure described in the proof, consider the linear threshold function:

$$3x_1 - 5x_2 + 6x_3 \geq 5.$$

Dealing with the negative weight and reordering of the terms yields

$$6x_3 + 5\bar{x}_2 + 3x_1 \geq 10.$$

Summarizing, we assume w.l.o.g. that the input to the rule-extracting algorithms is a *canonical linear threshold function*

**Definition 10.** *A linear threshold function $t(\mathbf{x}) = H(w_1x_1 + \cdots + w_nx_n - \theta)$ is in canonical form if $w_1 \geq w_2 \geq \cdots \geq w_n > 0$.*

**Algorithm 1** *ExtractNC* - Extracts a Nested Canalyzing Function, if possible

**Input:** A canonical threshold function $t \equiv w_i x_i + \cdots + w_n x_n \geq \theta$, $1 \leq i \leq n$.
**Output:** An NC-function equivalent to $t$ if there is one, else NONE.

$ExtractNC(w_i x_i + \cdots + w_n x_n \geq \theta)$:
**if** $\theta \leq 0$ **then**
  return 1;
**else if** $\sum_{j=i}^{n} w_j < \theta$ **then**
  return 0;
**end if**/* $0 < \theta \leq \sum_{j=i}^{n} w_j$ */
**if** $i = n$ **then**
  return $x_n$;
**end if**; /* $i < n$ and $0 < \theta \leq \sum_{j=i}^{n} w_j$ */
**if** $w_i \geq \theta$ **then**
  $g(x_{i+1}, \ldots, x_n) \leftarrow ExtractNC(w_{i+1} x_{i+1} + \cdots + w_n x_n \geq \theta)$;
  return $x_i \vee g(x_{i+1}, \cdots, x_n)$;
**else if** $\sum_{j=i+1}^{n} w_j < \theta$ **then**
  $g(x_{i+1}, \ldots, x_n) \leftarrow ExtractNC(w_{i+1} x_{i+1} + \cdots + w_n x_n \geq \theta - w_i)$;
  return $x_i \wedge g(x_{i+1}, \cdots, x_n)$;
**else**
  return NONE;
**end if**

Note that 1, 0 and NONE satisfy $1 \vee g(\cdots) = 1$, $1 \wedge g(\cdots) = g(\cdots)$, $0 \wedge g(\cdots) = 0$, $0 \vee g(\cdots) = g(\cdots)$, NONE $\wedge\, g(\cdots)$ = NONE, and NONE $\vee\, g(\cdots)$ = NONE.

**Theorem 11.** *$ExtractNC$ takes polynomial time to extract a nested canalyzing function from a given canonical threshold function if possible, outputting NONE if there is no such function.*

*Proof.* It is straightforward to see that $ExtractNC$ works in polynomial time. We prove the correctness of $ExtractNC$ by induction on the number of variables.

Clearly $ExtractNC$ returns a correct nested canalyzing function when the threshold function consists of a single variable. For the induction step suppose $i$ is given and that $ExtractNC$ returns a correct solution (possibly NONE) given any canonical threshold function of the form $w'_{i+1} x_{i+1} + \cdots + w'_n x_n \geq \theta$. We will show that then a correct solution is returned for any canonical threshold function $w_i x_i + \cdots + w_n x_n \geq \theta$.

When $w_i \geq \theta$, the threshold function has value 1 on an assignment $(a_i, \ldots, a_n)$ to $(x_1, \ldots, x_n)$, i.e. $w_i a_i + \cdots + w_n a_n \geq \theta$, if and only if either $a_i = 1$ or both $a_i = 0$ and $w_{i+1} a_{i+1} + \cdots + w_n a_n \geq \theta$. Therefore, $w_i x_i + \cdots + w_n x_n \geq \theta$ corresponds to a nested canalyzing function iff $w_{i+1} x_{i+1} + \cdots + w_n x_n \geq \theta$ corresponds to a nested canalyzing function $g(x_{i+1}, \ldots, x_n)$, and if so it corresponds to $x_i \vee g(x_{i+1}, \ldots, x_n)$.

In case $\sum_{j=i+1}^{n} w_j < \theta$ the threshold function has value 1 on an assignment $(a_i, \ldots, a_n)$ if and only if $a_i = 1$ and $w_{i+1} a_{i+1} + \cdots + w_n a_n \geq \theta - w_i$. Thus $x_i \wedge g(x_{i+1}, \ldots, x_n)$ is the nested canalyzing function corresponding to the threshold function, where $g(x_{i+1}, \ldots, x_n)$ corresponds to $w_{i+1} x_{i+1} + \cdots + w_n x_n \geq \theta - w_i$.

To prove that the the algorithm is correct also when it returns NONE, we prove that if there is a nested canalyzing function $f$ that is equivalent to the canonical threshold function $t(\mathbf{x}) \equiv w_i x_i + \cdots + w_n x_n \geq \theta)$ then either $w_i \geq \theta$ or $\sum_{j=i+1}^{n} w_j < \theta$. As stated in the algorithm we can assume that $i < n$ and

$$0 < \theta < \sum_{j=i}^{n} w_j. \tag{1}$$

Suppose first that $f$ has the form $f(x_i, \ldots, x_n) = \ell_k \vee g(x_i, \ldots, x_n)$, for some $i \leq k \leq n$, and $g$ not a function of $x_k$. Although $\ell_k$ could in general be $x_k$ or $\overline{x_k}$, here $\ell_k = \overline{x_k}$ is impossible: otherwise $f(\mathbf{0}) = 1$, whereas $t(\mathbf{0}) = 0$ since $0 < \theta$.

Thus $f = 1$ whenever $x_k = 1$. Consider the assignment $\mathbf{a}$ with values $a_k = 1$, and $a_j = 0$ for $j \neq k$. Then from $t(\mathbf{a}) = 1$ it follows that $w_k \geq \theta$.

If $f$ is of the form $f(\mathbf{x}) = x_k \wedge g$ then $f(\mathbf{a}) = 0$ for any assignment $\mathbf{a}$ such that $a_k = 0$. Arguing analogously to the above it follows that $\sum_{j \neq k} w_j < \theta$.

Finally, we prove that the index $k$ appearing in the argument above can in fact be taken to be $i$, because $t$ is in canonical form. Consider, for example, the case that $f$ has the form $x_k \vee g(x_i, \ldots, x_n)$, with $i \neq k$ and $g(x_i, \ldots, x_n)$ not dependent on $x_k$. We saw that then $w_k \geq \theta$. From $w_i \geq w_k$ it follows that $w_i + w_{i+1}a_{i+1} + \cdots + w_n a_n \geq \theta$ for any $a_{i+1}, \ldots, a_n$, i.e. $t(\mathbf{a}) = 1$, whenever $a_i = 1$. Thus in fact $f = x_i \vee x_k \vee g'$.

Similarly, if $f$ has the form $x_k \wedge g(x_i, \ldots, x_n)$ then $\sum_{j>i} w_j \leq \sum_{j \neq k} w_j < \theta$, so that $t(\mathbf{a}) = 0$ whenever $a_i = 0$. Consequently $f = x_i \wedge x_k \wedge g'$. □

Next we present an algorithm to test whether a canonical threshold function is equivalent to the $k$-out-of-$n$ majority function, $Maj_k(x_1, \ldots, x_n)$. Although several methods have been developed for the simplification of a linear threshold function into a set of majority functions (Barth, 1993), it is unclear whether any of these methods is directly applicable to the extraction of a majority function, and our algorithm has the virtue of being very simple.

**Algorithm 2** *ExtractMF* - Extracts a Majority Function if possible

**Input:** A canonical threshold function $t$ with weights $w_1 \geq \cdots \geq w_n > 0$.
**Output:** $Maj_k(x_1, \ldots, x_n)$, $k > 1$, if it is equivalent to $t$, else NONE.

```
ExtractMF(w_1 x_1 + ... + w_n x_n ≥ θ):
for k = 2 to n do
  if w_{n-k+1} + ... + w_n ≥ θ and w_1 + ... + w_{k-1} < θ then
    return Maj_k(x_1, ..., x_n);
  end if
end for
return NONE;
```

**Proposition 12.** *ExtractMF returns the correct answer in polynomial time.*

*Proof.* The algorithm clearly takes polynomial time, computing no more than $n$ times two sums of no more than $n$ terms. To prove correctness assume first that $t$ does correspond to $Maj_k$ for some $k > 1$. Since $Maj_k(x_1, \ldots, x_n)$ is the disjunction of all conjunctions consisting of $k$ variables it has the value 1 for the assignment $a_1 = \cdots = a_{n-k} = 0$, $a_{n-k+1} = \cdots = a_n = 1$, meaning that $w_{n-k+1} + \cdots + w_n \geq \theta$. But it has the value 0 for the assignment $a_1 = \cdots = a_{k-1} = 1$, $a_k = \cdots = a_n = 0$, as each conjunction contains at least one 0 multiplicand. Since $t(\mathbf{a}) = 0$ means that $w_1 + \cdots + w_{k-1} < \theta$, this $Maj_k$ will be output by *ExtractMF*.

Conversely, assume that *ExtractMF* returns $Maj_k$, so that $w_{n-k+1} + \cdots + w_n \geq \theta$. Because $w_1 \geq \cdots \geq w_n > 0$ this implies that $\sum_{j=1}^{k} w_{i_j} \geq \theta$ for any $1 \leq i_1 \leq \cdots \leq i_k \leq n$, i.e. that $t(\mathbf{a}) = 1$ for any assignment $\mathbf{a}$ that contains at least $k$ 1's. Furthermore, $w_1 + w_2 + \cdots + w_{k-1} < \theta$ also holds for this $k$, which similarly implies that $t(\mathbf{a}) = 0$ for any assignment $\mathbf{a}$ that contains at most $k - 1$ 1's. Thus, the input canonical threshold function is equivalent to $MF_k$. □

### 3.2 Extraction of a general Boolean function

If the given canonical threshold function is equivalent to a nested canalyzing function or a $k$-out-of-$n$ majority function then the algorithms of the previous section are able to construct a concise rule. What to do if the algorithms are unsuccessful is the question we address in this section. The basic idea is to break the threshold function up into two threshold functions, using the following equivalence, and to recurse :

$$\begin{aligned} w_1 x_1 + \cdots + w_n x_n \geq \theta \quad &\Longleftrightarrow \\ (x_1 \wedge (w_2 x_2 + \cdots + w_n x_n \geq \theta - w_1)) \quad &\vee \quad (\bar{x}_1 \wedge (w_2 x_2 + \cdots + w_n x_n \geq \theta)). \end{aligned}$$

In order to avoid as much as possible the potentially exponential size of the resulting Boolean function, the Algorithm tries at each stage to extract a concise Boolean function using Algorithm 1 and Algorithm 2.

**Algorithm 3** ExtractBF - extracts a Boolean Function

**Input:** A canonical threshold function.
**Output:** A Boolean function.
$ExtractBF(w_i x_i + \cdots + w_n x_n \geq \theta)$:
$g \leftarrow ExtractNC(w_i x_i + \cdots + w_n x_n \geq \theta)$;
**if** $g$ is not NONE **then**
return $g$;
**end if**
$g \leftarrow$ *ExtractMF*$(w_i x_i + \cdots + w_n x_n \geq \theta)$;
**if** $g$ is not NONE **then**
return $g$;
**end if**
return $(x_i \wedge ExtractBF(w_{i+1} x_{i+1} + \cdots + w_n x_n \geq \theta - w_i)) \vee ExtractBF(w_{i+1} x_{i+1} + \cdots + w_n x_n \geq \theta)$;

**Proposition 13.** *ExtractBF transforms any linear threshold function into a Boolean function composed of disjunctions, conjunctions, and majority functions of literals in $O^*(2^n)$ time.*

In the worst case, the algorithm may output an exponential size Boolean function using exponential time. However, the algorithm works in polynomial time with respect to the output size, and the output size is expected to be small if the given threshold function can be represented as a composition of majority functions, nested canalyzing functions, disjunctions and conjunctions.

Let us briefly compare our algorithm with Tsukimoto's algorithm (Tsukimoto, 2000), which outputs the DNF form of a Boolean function, and works in polynomial time if the size of each term is bounded by a constant. Otherwise it is an exponential time algorithm, or an approximate one. Our algorithm is exact and works in polynomial time if a given linear threshold function corresponds to a nested canalyzing function or a $k$-out-of-$n$ majority function.

In all cases that we checked our algorithm outputs a Boolean function that is more concise than the one output by Tsukimoto's algorithm. For example, given a linear threshold function $0.5x_1+0.4x_2+0.1x_3+0.1x_4+0.1x_5+0.1x_6 \geq 1$, our algorithm outputs $x_1x_2(x_3 \vee x_4 \vee x_5 \vee x_6)$, whereas Tsukimoto's algorithm outputs $x_1x_2x_3 \vee x_1x_2x_4 \vee x_1x_2x_5 \vee x_1x_2x_6$. For another example, given a linear threshold function $0.5x_1 + 0.5x_2 + 0.4x_3 + 0.4x_4 + 0.4x_5 \geq 0.8$, our algorithm outputs $Maj_2(x_1, \ldots, x_5)$. whereas Tsukimoto's algorithm outputs $x_1x_2 \vee x_1x_3 \vee x_1x_4 \vee x_1x_5 \vee x_2x_3 \vee x_2x_4 \vee x_2x_5 \vee x_3x_4 \vee x_3x_5 \vee x_4x_5$.

However, the rule extracted by our algorithm can be longer than necessary, even if the rule is a majority function but not on all $n$ variables. Consider the function whose weights are 12,11,11,10,10,10,1 and its threshold is $\theta = 29$. This function is equivalent to $Maj_3(x_1, \ldots, x_6)$, but our algorithm outputs

$$\begin{aligned}
&(x_1 \wedge ((x_2 \wedge (x_3 \vee x_4 \vee x_5 \vee x_6)) \vee (x_3 \wedge (x_4 \vee x_5 \vee x_6)) \vee (x_4 \wedge (x_5 \vee x_6)) \\
&\vee(x_5 \wedge x_6))) \\
&\quad \vee(x_2 \wedge ((x_3 \wedge (x_4 \vee x_5 \vee x_6)) \vee (x_4 \wedge (x_5 \vee x_6)) \vee (x_5 \wedge x_6))) \\
&\quad \vee(x_3 \wedge ((x_4 \wedge (x_5 \vee x_6)) \vee (x_5 \wedge x_6))) \\
&\quad \vee(x_4 \wedge x_5 \wedge x_6).
\end{aligned}$$

If it is important to extract a concise (and therefore more informative) rule, even if it may take considerably more time, one can modify the above algorithm by replacing its last line by the following:

$$\begin{aligned}
&g_1 \leftarrow (x_i \wedge ExtractBF(w_{i+1} x_{i+1} + \cdots + w_n x_n \geq \theta - w_i)) \\
&\quad \vee ExtractBF(w_i x_i + \cdots + w_{n-1} x_{n-1} \geq \theta); \\
&g_2 \leftarrow (x_n \wedge ExtractBF(w_i x_i + \cdots + w_{n-1} x_{n-1} \geq \theta - w_n)) \\
&\quad \vee ExtractBF(w_{i+1} x_{i+1} + \cdots + w_{n-1} x_{n-1} \geq \theta); \\
&\textbf{if } |g_1| \leq |g_2| \text{ return } g_1 \textbf{ else } return\ g_2;
\end{aligned}$$

In the above, we considered the problem of extracting a Boolean function from a single neuron. We consider next a network with hidden layers. In this case the extraction of the Boolean function corresponding to a given neuron involves combining the Boolean functions extracted for all its input neurons, which may result in a very complicated function.

We propose two methods for alleviating this problem. The first one is to add constraints on the length of the generated Boolean functions. Computational experiments using this approach are presented in Section 5.1. The second one is

to extract rules that are conditional probabilities, rather than deterministic ones. In the next section we describe this probabilistic rule extraction algorithm.

# 4 Extraction of Probabilistic Rules from a Neural Network

## 4.1 Problem Definition

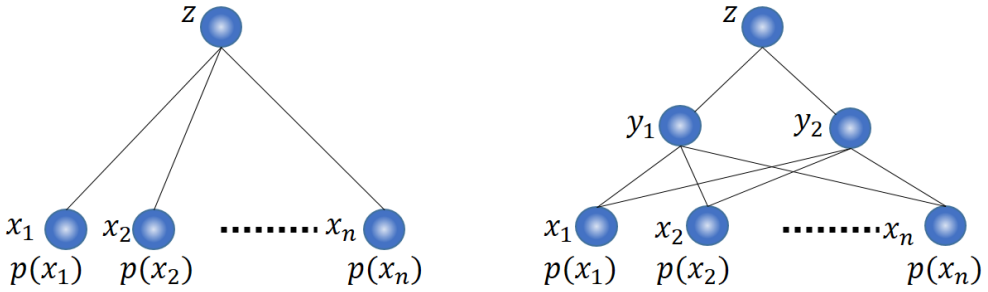


Figure 2: A network with two layers (left) and a network with one hidden layer (right).

In this section, we consider the problem of extracting conditional probabilities given a combination of some input nodes values. Suppose that $z$ is the output neuron directly connected to $n$ input neurons $x_1, \ldots, x_n$ (see Fig. 2, left). We assume that the input neurons obey independent distributions, meaning that the value $Pr(x_i = 1)$ depends only on $i$. We also assume that the representation of $Pr(x_i = 1)$ uses only $O(1)$ bits, as is the case for the uniform distribution $Pr(x_i = 1) = 0.5$. The *Probabilistic Rule Extraction* problem is to compute the probabilities $Pr(z = a)$, $Pr(z = a|x_i = b)$, $Pr(z = a|x_{i_1} = b_1, x_{i_2} = b_2)$, $\cdots$ for $a, b, b_i \in \{0, 1\}$, up to a fixed number of input nodes. The problem can be extended to neural networks having hidden layers (Fig. 2, right).

## 4.2 NP-hardness

We prove that probabilistic rule extraction is NP-hard, even for a neural network with one hidden layer.

**Theorem 14.** *Deciding whether $Pr(z = 1) > 0$ is NP-hard for a neural network with one hidden layer.*

*Proof.* The proof consists of a polynomial time reduction from 3SAT (Garey and Johnson, 1979). Let $v_1, \ldots, v_N$ be the variables of the 3-SAT formula, and let $c_1, \ldots, c_L$ be its clauses, each clause being a logical OR of at most three literals, $c_i = \ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3}$ where $\ell_{i_j}$ is either $v_{i_j}$ or $\bar{v}_{i_j}$. 3SAT is the problem of deciding whether or not there exists an assignment of 0-1 values to $v_1, \ldots, v_N$ which satisfies all the clauses (i.e., all clauses evaluate to 1).

From an instance of 3SAT, we construct a neural network as follows (see also Fig. 3).

The input nodes are $X = \{x_1, \ldots, x_N\}$, and the hidden layer nodes are $Y = \{y_1, \ldots, y_L\}$, where we assume the uniform 0-1 distribution for each of the $x_i$. Node $y_i$ in the hidden layer corresponds to clause $c_i$, and its weights are $w_{i,i_j} = 1$ if $\ell_{i_j} = v_{i_j}$, and $w_{i,i_j} = -1$ if $\ell_{i_j}$ is $\bar{v}_{i_j}$, with $w_{i,k} = 0$ for $k \neq i_1, i_2, i_3$. The threshold of node $y_i$ is $\theta_{y_i} = 1 - |\{\ell_{i_j} | \ell_{i_j} = \bar{x}_{i_j}\}|$. The weights assigned to node $z$ are $w_1 = w_2 = \cdots = w_L = 1$ and its threshold is $\theta = L$.

It is straightforward to verify that $Pr(z = 1) > 0$ iff there exists an assignment such that all $c_1, \ldots, c_L$ evaluate to 1. The reduction clearly takes polynomial time. □

In this reduction the number of nodes in the hidden layer is not bounded, whereas the weights are restricted to be -1, 0, and 1. The next Theorem points out that the problem remains NP-hard even if the number of nodes in the hidden layer is restricted to be at most 2, provided the weights can be arbitrary integers.

**Theorem 15.** *Deciding whether $Pr(z = 1) > 0$ is NP-hard for a neural network with one hidden layer with two neurons if $O(n)$ bits integer weights can be assigned.*

*Proof.* As in the proof of the co-NP hardness of testing the equivalence of two threshold functions (Melkman et al., 2018), we use a polynomial-time reduction from the subset sum problem. Recall that the subset sum problem asks: given a set of positive integers $A = \{a_1, a_2, \ldots, a_N\}$ and an integer $b$, is there a 0-1 assignment $(v_1, \ldots, v_N)$ such that $a_1 v_1 + a_2 v_2 + \cdots + a_N v_N = b$?

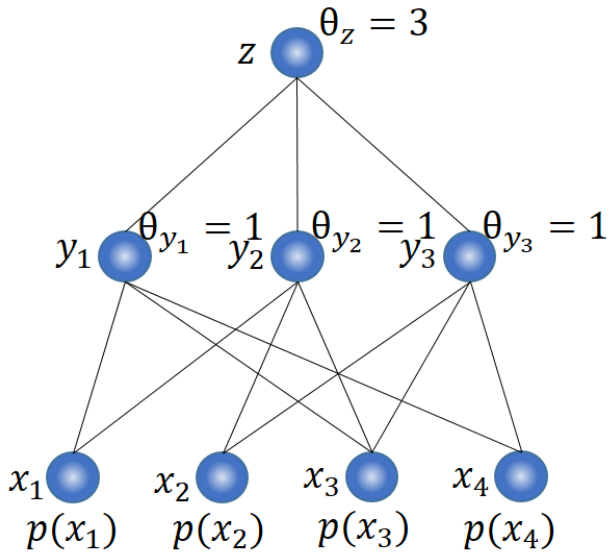


Figure 3: Reduction from a 3-SAT instance $\{v_1 \vee v_3 \vee v_4, v_1 \vee v_2 \vee v_3, v_2 \vee v_3 \vee v_4\}$ to probabilistic rule extraction.

From an instance $(A, b)$ of the subset sum problem, we construct a neural network similar to the one used in the proof of Theorem 14, with only two nodes $y_1$ and $y_2$ in the hidden layer. The weights associated with these two nodes are $w_{i,j} = a_j,\ i = 1, 2,\ j = 1, \ldots, N$, and their thresholds are $\theta_1 = b$, and $\theta_2 = b + 1$. The weights of the output node $z$ are $w_1 = 1$ and $w_2 = -1$, and $\theta_z = 1$.

Clearly $Pr(z = 1) > 0$ iff there exists a 0-1 assignment $(v_1, \ldots, v_N)$ such that $a_1v_1 + a_2v_2 + \cdots + a_Nv_N \geq b$ and $a_1v_1 + a_2v_2 + \cdots + a_Nv_N < b + 1$. Furthermore, the time taken by the reduction is polynomial in the total number of bits used to represent the instance of the Subset Sum problem. □

### 4.3 Probabilistic Rule Extraction from a Linear Threshold Function

Let us turn now to developing an algorithm for calculating $Pr\ (z = 1)$ given a linear threshold function. As a first step consider the case that the threshold function is a nested canalyzing function.

Let *ncf*($\mathbf{x}$) be a nested canalyzing function on the variables $\mathbf{x}$ and let $p_i$ denote the probability

$$p_i = Pr(x_i = 1). \tag{2}$$

We assume that a constant number of bits suffices to specify $p_i$. According to Lemma 9 we can assume that the threshold function is in canonical form, and therefore its nested canalyzing form contains no negated variables, and the variables appear ordered by non-increasing weights (cf. Algorithm 1). Here is the pseudo-code for calculating $Pr(ncf(\mathbf{x}) = 1)$.

**Algorithm 4** Probability Extraction Algorithm for a nested canalyzing function

**Input:** A canonical threshold function in the form of a nested canalyzing function $ncf(x_i, \cdots, x_n)$; probabilities $p_j, j = i, \ldots, n$;
**Output:** $Pr(ncf(x_i, \ldots, x_n) = 1)$;
*ExtractP*($ncf(x_i, \cdots, x_n)$):
**if** $i = n$ **then** return $p_n = 1$;
**end if**
**if** $ncf(x_i, \cdots, x_n) = x_i \wedge g(x_{i+1}, \cdots, x_n)$ **then**
return $p_i * ExtractP(g(x_{i+1}, \cdots, x_n))$;
**else if** $ncf(x_i, \cdots, x_n) = x_i \vee g(x_{i+1}, \cdots, x_n)$ **then**
return $p_i + (1 - p_i) * ExtractP(g(x_{i+1}, \cdots, x_n))$;
**end if**

**Proposition 16.** *A probabilistic rule can be extracted from a nested canalyzing function in polynomial time.*

*Proof.* The correctness of Algorithm 4 is self-evident. Since by assumption the probabilities $p_i$ are represented with a constant number of bits, each call makes at most one multiplication, and the number of calls is linear in $n$, the algorithm works in time polynomial in $n$. □

Note that $Pr(z = 1|x_{i_1} = a_{i_1}, \ldots, x_{i_k} = a_{i_k})$ can be computed by applying Algorithm 4 after setting $(x_{i_1}, \ldots, x_{i_k})$ to $(a_{i_1}, \ldots, a_{i_k})$.

Next, we present an algorithm for a general canonical threshold function. It is based on the following straightforward recursion.

$$Pr(f(x_1, \ldots, x_n) = 1) = p_1 Pr(f(x_1, \ldots, x_n) = 1|x_1 = 1) + (1 - p_1))Pr(f(x_1, \ldots, x_n) = 1|x_1 = 0). \quad (3)$$

To implement the recursion efficiently we use dynamic programming, as in (Melkman et al., 2018). Given a canonical threshold function $w_1x_1 + \cdots + w_nx_n \geq \theta$ with positive integer weights $w_i$ and threshold $\theta$, define $T(i, s)$ by

$$T(i, s) = Pr(\sum_{j=i}^{n} w_j x_j \geq s).$$

From recursion 3 we get a recursion for $T(i, s)$:

$$T(i, s) = p_i T(i + 1, s - w_i) + (1 - p_i)T(i + 1, s). \quad (4)$$

Note that the value we are after is $Pr(z = 1) = T(1, \theta)$; it is obtained by computing recursively, with memorization, the values $T(i, s),\ i = 1, \ldots, n$, for appropriate values of $s$. Since all weights are positive, the smallest and largest values of $s$ of possible interest are 0 and $\theta$. The table is initialized by $T(i, s) = 1$ if $s \leq 0$ and

$$T(n, s) = 1 \text{ if } s \leq 0,\ T(n, s) = p_n \text{ if } 0 < s \leq w_n,\ T(n, s) = 0 \text{ if } \ w_n < s. \quad (5)$$

**Theorem 17.** *Given a canonical threshold function the value $Pr(z = 1)$ can be computed in $O^*(n\theta)$ time.*

*Proof.* The correctness of the dynamic programming procedure is self evident.

The number of table entries $T(i, s)$ needed for the computation is $n\theta$. The size of each entry is polynomially bounded, so that the computations of Equations (4) and (5) take polynomial time. □

Since $\theta$ may be exponentially large in terms of the input size, this algorithm runs in pseudo-polynomial time, although if $\theta$ is polynomially large then so is the running time.

Here is an example illustrating the working of the algorithm. Suppose that we are requested to compute $Pr(2x_1 + 3x_2 + x_3 - 2x_4 \geq 2)$ where $Pr(x_i = 1) = 0.5$ for $i = 1, \ldots, 4$. The canonical form of the threshold function is $3x_1 + 2x_2 + 2x_3 + x_4 \geq 4$. According to Equation (5) the initial values are

$$T(4, s) = 1 \text{ if } s \leq 0,\ \frac{1}{2} \text{ if } 0 < s \leq 1,\ 0 \text{ if } 1 < s.$$

A straightforward computation using Equation (4) yields

$$Pr(2x_1 + 3x_2 + x_3 - 2x_4 \geq 2) = T(1, 4) = p_1 T(2, 1) + (1 - p_1)T(2, 4) = \cdots = \frac{9}{16}.$$

Only 8 values were computed, half of the table.

### 4.4 Probabilistic Rule Extraction: Network with One Hidden Layer

Here we extend the dynamic programming algorithm presented above to a neural network with one hidden layer; the next subsection describes how to handle multiple hidden layers. In these subsections $n_i$ denotes the number of input nodes in layer $i$, with $n = n_1$ being the number of input nodes. The right side of Fig. 2 shows the structure of a network with one hidden layer.

**Caution:** In these two subsections we consider two or more functions simultaneously. Therefore we do not assume that all threshold functions are given in canonical form. Using the method of Lemma 9 we can still assume that the weights of the threshold functions are all non-negative; the price we pay is that the threshold functions are combinations of *literals* rather than of variables. Given an assignment $\mathbf{a}$ to the variables $\mathbf{x}$ of a threshold function $w_1\ell_1 + \cdots + w_n\ell_n \geq$

$\Gamma$, we will denote by $\ell_j(a_j)$ the value given to $\ell_j$ by the assignment $x_j = a_j$; for example, if $\ell_j = \overline{x_j}$ and $a_j = 1$ then $\ell_j(a_j) = 0$.

Consider to begin with the simplest case of a neural network having one hidden layer with two nodes $y_1$ and $y_2$. The values of $y_1$ and $y_2$ are determined by the threshold functions

$$w_{k,1}\ell_{k,1} + \cdots + w_{k,n}\ell_{k,n} \geq \Gamma_k, k = 1, 2, \tag{6}$$

with $w_{k,j} \geq 0, k = 1, 2, 1 \leq j \leq n$. The value of the output node $z$ is 1 iff $w_{y_1}\ell_{y_1} + w_{y_2}\ell_{y_2} \geq T$ with $w_{y_1}, w_{y_2} \geq 0$, and $0 < T < w_{2,1} + w_{2,2}$ (otherwise the value is constant). For simplicity we will assume in the following that $\ell_{y_k} = y_k, k = 1, 2$; the other possibilities are handled similarly.

We describe next two ways to compute the probability $Pr(z = 1)$.

1. The first method starts by initializing $P_3(1)$ to 0. Then it considers all assignments $\mathbf{a}$ to the variables of the input layer, and for each it computes
   (a) the resulting assignment $(b_1, b_2)$ to $(y_1, y_2)$:
   $$b_k = (w_{k,1}\ell_{k,1}(\mathbf{a}) + \cdots + w_{k,n}\ell_{k,n}(\mathbf{a}) \geq \Gamma_k), k = 1, 2;$$
   (b) the resulting value of $z$: $c = (w_{y_1}\ell_{y_1}(b_1) + w_{y_2}\ell_{y_2}(b_2) \geq T)$;
   (c) adds $Pr(\mathbf{x} = \mathbf{a})$ to $P_3(1)$ if $c = 1$.

   Since the input node variables are independent $Pr(\mathbf{x} = \mathbf{a}) = \prod_{j=1}^{n} Pr(x_j = a_j)$. $Pr(z = 1)$ equals the final value of $P_3(1)$. The total computation time is $O(n2^n)$.
   This computation is easily generalized to a network with $n_1$ input nodes and a hidden layer containing $n_2$ nodes $\mathbf{y}$: (1a) computes the assignment $\mathbf{b}$ to $\mathbf{y}$ in time $O(n_1 n_2)$, and (1b) computes $c = \left(\sum_{j=1}^{n_2} w_{y_j}\ell_{y_j}(b_j) \geq T\right)$. Thus the computation time is
   $$O(n_1 n_2 2^{n_1}). \tag{7}$$

2. The basic idea of the second method is to compute the joint probabilities of the two nodes in the hidden second layer, $P_2(b_1, b_2) = Pr(y_1 = b_1, y_2 = b_2)$, $b_1, b_2 \in \{0, 1\}$. With these values in hand the probability that $z = 1$ is
   $$Pr(z = 1) = H(w_{y_1} - T)P_2(1, 0) + H(w_{y_2} - T)P_2(0, 1) + P_2(1, 1). \tag{8}$$
   The method first computes the matrix $F(i, s_1, s_2),\ 1 \leq i \leq n,\ 0 \leq s_k \leq \Gamma_k$, defined by
   $$F(i, s_1, s_2) = Pr(\sum_{j=i}^{n} w_{1,j}\ell_{1,j} \geq s_1 \text{ and } \sum_{j=i}^{n} w_{2,j}\ell_{2,j} \geq s_2).$$
   Once all of $F$ has been computed, the value of $P_2(1, 1)$ is given by $P_2(1, 1) = F(1, \Gamma_1, \Gamma_2)$. Although the other values needed for Equation 8, $P_2(1, 0)$ and $P_2(0, 1)$, do not themselves appear in $F$, they can be derived from values that do appear: $P_2(1, *) = Pr(y_1 = 1) = P_2(1, 1) + P_2(1, 0) = F(1, \Gamma_1, 0)$, and $P_2(*, 1) = F(1, 0, \Gamma_2)$. Note that the computation accessed all values $P_2(c_1, c_2)$, $c_1, c_2 \in \{0, 1, *\}$.
   $F$ can be computed using a recursion similar to the one for $T$.
   $$\begin{aligned} F(i, s_1, s_2) = {} & p_i F(i + 1, s_1 - \ell_{1,i}(1)w_{1,i}, s_2 - \ell_{2,i}(1)w_{2,i}) \\ & + (1 - p_i)F(i + 1, s_1 - \ell_{1,i}(0)w_{1,i}, s_2 - \ell_{2,i}(0)w_{2,i}), \end{aligned} \tag{9}$$
   with the initialization
   $$\begin{aligned} F(n, s_1, s_2) = {} & p_n H((\ell_{1,n}(1)w_{1,n} - s_1)H((\ell_{2,n}(1)w_{2,n} - s_2) \\ & + (1 - p_n)H((\ell_{1,n}(0) - s_1)H((\ell_{2,n}(0) - s_2). \end{aligned} \tag{10}$$
   In particular, $F(n, s_1, s_2) = p_n H(\ell_{1,n}(1)w_{1,n} - s_1) + (1 - p_n)H(\ell_{1,n}(0) - s_1)$ for $s_2 \leq 0$, and $F(n, s_1, s_2) = 1$ when both $s_1$ and $s_2$ are non-positive.
   The time complexity of the resulting algorithm is pseudo-polynomial, $O^*(n\Gamma_1\Gamma_2)$.
   To extend these ideas to a neural network with a single hidden layer containing $n_2$ nodes, define the matrix $F(i, s_1, \ldots, s_{n_2})$; it can be computed recursively using the analogs of Equations (9) and (10). From this matrix we derive the values $P_2(\mathbf{c})$ with $\mathbf{c} \in \{0, 1, *\}^{n_2}$ in $n_2$ stages, where at stage $r$ we consider those

$\mathbf{c}$ containing exactly $r$ 0's. For such a $\mathbf{c}$ let $j_r$ be the first index such that $c_{j_r} = 0$, and define $\mathbf{t}$ and $\mathbf{u}$ by $t_j = u_j = c_j$ if $j \neq j_r, 1 \leq j \leq n_2$ and $t_{j_r} = *, u_{j_r} = 1$. Then $P_2(\mathbf{c}) = P_2(\mathbf{t}) - P_2(\mathbf{u})$. Each such computation subtracts one already computed value from another and takes constant time. The initialization stage of the computation is to copy the values of $P_2(\mathbf{c})$ for $\mathbf{c} \in \{1, *\}^{n_2}$ from $F$: $P_2(\mathbf{c}) = F(1, \mathbf{t})$, where $t_j = \Gamma_j$ if $c_j = 1, t_j = 0$ if $c_j = *, 1 \leq j \leq n_2$.

On completing the computation of $P_2$ from $F$ we know all $3^{n_2}$ values $P(\mathbf{c})$ with $\mathbf{c} \in \{*, 0, 1\}^{n_2}$. Consequently the time to compute all of $P_2$ is $O(3^{n_2})$, and the overall computation time for this method is $O(n_1 \prod_{k=1}^{n_2} \Gamma_k + 3^{n_2})$. Thus this method is much faster than the first naive method, Equation (7), if $n_1 \gg \max\{\log_2 3 \cdot n_2, \sum_{k=1}^{n_2} \log_2 \Gamma_k\}$.

### 4.5 Probabilistic Rule Extraction: Network with Multiple Hidden Layers

To extend the approach of the previous subsection further to multilayer neural networks, we assume that our computation model is the real RAM (random access machine) (Shamos, 1978). The real RAM is an idealized model in which it is assumed that any real number can be stored in a word and that each arithmetic operation takes constant time. This model has been widely used for analyzing algorithms in computational geometry (Preparata and Shamos, 1985). In actual computers, real numbers are of course represented as finite precision floating point numbers. Therefore, when implemented on actual computers, our algorithms may output approximate probabilities.

Consider a neural network with $L$ ($L \geq 4$) layers, the first one being the input layer, the last one consisting of the lone output node $z$, and the $i$th layer having $n_i$ nodes for $i = 1, \ldots, L$. We briefly indicate how to generalize the algorithms of the previous sections to the current setting.

The first generalized method computes, for each assignment $\mathbf{a}$ to the variables of the input layer, the assignment generated to the variables of the $i+1$-st layer by the assignment that was generated to the variables of the $i$-th layer, $i = 1, \ldots, L-1$. If the resulting assignment to $z$ is 1, then $Pr(\mathbf{a})$ is added to the current value of $P_L(1)$. At the completion of the computation $Pr(z = 1) = P_L(1)$. The total time taken by the computation is $O(2^{n_1} \sum_{i=1}^{L-1} n_i n_{i+1})$.

The second generalized method is a combination of the previous two methods: it begins by applying the second method to compute all joint probabilities for the first hidden layer, Next it essentially uses the first generalized method just described, but starts it from the second layer. Thus, for each assignment $\mathbf{b}$, whose probability $Pr(\mathbf{b})$ was just calculated, the method traces the assignment it generates to the variables of the $i$-th layer, $i = 3, \ldots, L-1$. If the resulting assignment to $z$ is 1, then $Pr(\mathbf{b})$ is added to the current value of $P_L(1)$. Upon termination this method returns $P_L(1)$ as the value of $Pr(z = 1)$. Thus the running time is

$$O(n_1 \prod_{k=1}^{n_2} \Gamma_k + 3^{n_2} + 2^{n_2} \sum_{i=2}^{L-1} n_i n_{i+1}). \tag{11}$$

Then the following Theorem summarizes the foregoing description.

**Theorem 18.** *Given a neural network with $L$ ($L \geq 4$) layers, suppose $n_1 \geq \max\{\log_2 3 \cdot n_2, \sum_{k=1}^{n_2} \log_2 \Gamma_k\}$. Then $Pr(z = 1)$ can be computed in time $O(n_1 \prod_{k=1}^{n_2} \Gamma_k + 3^{n_2} + 2^{n_2} \sum_{i=2}^{L-1} n_i n_{i+1})$.*

Theorem 11 means that the second method is at least as fast as the first naive enumeration based method if $n_1 \geq \max\{\log_2 3 \cdot n_2, \sum_{k=1}^{n_2} \log_2 \Gamma_k\}$ and is much faster if $n_1 \gg \max\{\log_2 3 \cdot n_2, \sum_{k=1}^{n_2} \log_2 \Gamma_k\}$. Although many neurons are used in hidden layers in recent applications, there are several practical cases in which only a few neurons are used in hidden layers (Zhao and Thorpe, 2000) (Mazurowski et al., 2008). In such cases, the second method might work much faster than the naive method. At least this result gives a theoretical advance over the native method.

The analyses above are all for binary classifications. It might be possible to extend this probabilistic rule extraction to multiclass classification. A possible way is to compute $P(z_1 = 1), P(z_2 = 1), \ldots, P(z_p = 1)$ separately for neural networks with $p$ outputs.

# 5 Computational Experiments

## 5.1 Computational Experiments with Boolean Rule Extraction

In addition to the theoretical analyses of the Boolean rules extraction, we performed computational experiments to demonstrate its potential usefulness.

Although the algorithms presented in Section 3 are applicable only to single neurons, they can be extended to neural networks with hidden layers, using a strategy similar to the one given in (Tsukimoto, 2000). Namely, starting with the first layer, and proceeding layer by layer, the Boolean functions for all neurons in a layer are computed. The Boolean function so obtained for the single neuron in the final layer is the function for the output .

To study the performance of our proposed method, different neural networks were trained on one synthetic dataset and two real datasets, all with sigmoid functions using the Adam optimizer in PyTorch version 1.0.0 (Paszke et al., 2017). Binary cross-entropy is selected as our loss function. Since we want to see how well our Boolean rules approximate the neural networks we used in each case all of the dataset after samples with missing attributes were removed. The training was terminated once the training loss was less than 0.02.

In order to obtain concise Boolean rules, we ceased the generation of terms at the third order, where a term is of order $i$ if it has the form $\prod_{k=1}^{i} x_k$. Therefore, the rules contain only first, second and third order terms. Technically this meant adding to Algorithms 1 and 3 an order parameter $r \geq 0$ which indicates the highest order to be generated by the algorithm. For example, the heading of Algorithm 1 was changed to $ExtractNC(w_i x_i + \cdots + w_n x_n \geq \theta;\ r)$, to its body we added another base case "**if** $r = 0$ return 0" as a first statement, and the order parameter was decreased as necessary in other statements. For simplicity in this part the procedure $ExtractMF$ was deleted from Algorithm 3. Here is an example of this order-capped Boolean function generation: the threshold function $5x_1 + 3x_2 + 3x_3 + 2x_4 + 2x_5 \geq 10$ with order parameter $r = 3$ yields the function $x_1 \wedge ((x_2 \wedge (x_3 \vee x_4 \vee x_5)) \vee (x_3 \wedge (x_4 \vee x_5)))$.

1. *Regulation of the Mammalian Cell Cycle*

   As mentioned in Section 2.2, many biologically important functions are reported to be nested canalyzing. For example, the following rule is used in a logical model of the mammalian cell cycle network (Fauré et al., 2006):

   $$UbcH10 : (\overline{Cdh1}) \vee (Cdh1 \wedge Ubc \wedge (Cdc20 \vee CycA \vee CycB))$$

   In order to examine whether our algorithm can reconstruct this rule from samples, we generated at random a dataset of 16384 samples with the 15 binary attributes $Cdh1$, $Ubc$, $Cdc20$, $CycA$, $CycB$ and 10 noise attributes. Classes were $UbcH10$ and $\overline{UbcH10}$.

   These data were then used to train each one of the three neural networks shown in Fig. 4.

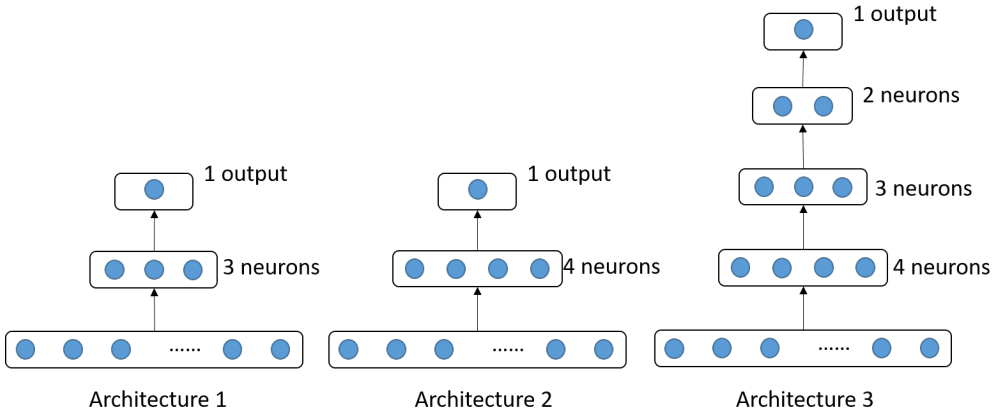


Figure 4: Architectures of the Neural Networks.

   Applying our Boolean rule extraction algorithm to the resulting trained neural networks yielded the correct nested canalyzing function for $UbcH10$ in each case.

2. *The Votes Data*

The votes data set consists of 232 out of the 435 voting records of congressmen in 1984, as they appear in the UCI Machine Learning Repository (Dua and Graff, 2017), after entries with missing data were removed. Each sample has 16 binary attributes (the votes of the congressman on 16 issues). Classes are $Democrat$ and $Republican$. Each extracted Boolean variable is of the form '$a$:$v$', meaning that $a$ has value $v$.

We trained each of the three neural networks depicted in Fig. 4 on these data. Then we tested our Boolean rule extraction algorithm on each of the resulting neural networks. In all three cases the following Boolean rule was obtained:

$$\begin{aligned} Democrat : (\text{physician-fee-freeze:}n) \vee ((&\text{adoption-of-the-budget-resolution:}y) \\ &\wedge (\text{anti-satellite-test-ban:}n) \\ &\wedge (\text{synfuels-corporation-cutback:}y)) \end{aligned}$$

The accuracy of the rule is $98.3\%$. This rule is the same as the one obtained for the $votes$ data by Tsukimoto (Tsukimoto, 2000) (with initial weight parameters 1).

3. *The Mushroom Data*

This dataset, also taken from the UCI Machine Learning Repository (Dua and Graff, 2017), consists of 5644 mushrooms (after removal of samples with missing values), each with 22 discrete physical characteristics. Classes are $edible$ and $poisonous$. The following Boolean rule was obtained for all three networks depicted in Fig. 4:

$$\begin{aligned} edible : \neg(\text{spore-print-color:green}) \wedge ((&\text{odor:none}) \\ &\vee (\text{odor:almond}) \\ &\vee (\text{odor:anise})) \end{aligned}$$

The accuracy of this Boolean rule is $99.7\%$.

The rule that Tsukimoto (Tsukimoto, 2000) extracted for the mushroom data is:

$$\begin{aligned} edible : (\text{gill-size:board}) \wedge ((&\text{odor:none}) \\ &\vee (\text{odor:almond}) \\ &\vee (\text{odor:anise})) \end{aligned}$$

and its $95.6\%$ accuracy for the 4062 mushrooms that Tsukimoto used is not as good as ours. Its accuracy on the full dataset of 5644 mushrooms is even slightly less, $94.5\%$.

Applying the C4.5 algorithm of (Quinlan, 1993) to our dataset yields the following rule:

$$(\text{odor:none}) \vee (\text{odor:almond}) \vee (\text{odor:anise}) \rightarrow edible$$

with an accuracy of $98.7\%$, slightly less than ours.

The best rule that extracted from the mushroom dataset is (Ishikawa, 2000):

$$\begin{aligned} edible : ((\text{gill-size:board}) \wedge \neg(\text{stalk-surface-below-ring:scaly}) \wedge \neg(\text{population:clustered}))\wedge \\ \neg(\text{spore-print-color:green}) \wedge ((&\text{odor:none}) \\ &\vee (\text{odor:almond}) \\ &\vee (\text{odor:anise})) \end{aligned}$$

The accuracy of this rule is $100\%$. Therefore, the method by Ishikawa (Ishikawa, 2000) might be useful in practice. However, there is no theoretical guarantee on what kinds of rules are extracted by his method. Therefore, we continued our experiments on our proposed methods.

We tested architecture 1 on this dataset using mean square error with different values. When the loss is $0.0228$, we obtained the same rule as the experiments using binary cross-entropy loss even if we ceased the generation of terms at the sixth order. We continued the training until the loss achieved $0.00258$. We got the same rule as previous. However, when the loss achieved $0.00086$, we obtained a rule with more than 20 literals even we ceased the generation of terms at the third order. These experiments indicate that use of very

small loss values might lead to overfitting and generation of complex rules. Therefore, the stopping condition should be carefully chosen in practice.

The above experiments are all focus on binary classifications. It might be possible to extend this Boolean rule extraction to multiclass classifications. One way is to consider neurons connected to each output as individual neural networks. Then Boolean rules can be obtained for each output.

### 5.2 Computational Experiments with Probabilistic Rule Extraction

This subsection has three parts. First, we examine the differences between the rules extracted by our algorithm as compared with the rules extracted directly from the trained sigmoid network. Second, we demonstrate that our algorithm can extract probabilistic rules with very small probabilities through experiment. Finally, we study the possibility of extracting important attributes from trained neural networks.

The probabilistic rule extraction algorithm outputs accurate probabilities for neural networks consisting of linear threshold functions. However, in practice sigmoid functions are often used in neural networks in order to enable training with sample data. For such networks, our algorithm outputs approximate probabilities. We report here on computational experiments aimed at examining the differences between the rules extracted by our algorithm as compared with the rules extracted directly from the trained sigmoid network.

In the experiments we trained a neural network consisting of ten input neurons, one output neuron, and one hidden layer with two neurons, all with sigmoid activation functions using the Adam optimizer in PyTorch version 1.0.0 (Paszke et al., 2017). Three sets of training samples with 500, 1024, 1024 samples respectively, ten attributes and one output label (all binary valued), were randomly generated from the following three Boolean functions:

1. 5-out-of-10 majority function ($Maj_5(x_1, \ldots, x_{10})$),
2. conjunction of two majority functions ($Maj_2(x_1, \ldots, x_5) \vee Maj_2(x_6, \ldots, x_{10})$),
3. $Maj_2(x_1, \ldots, x_5) \vee (x_6 \vee (x_7 \wedge (x_8 \vee (x_9 \wedge x_{10}))))$,

where we used $Pr(x_i = 1) = 0.5$ for all $i = 1, \ldots, n$. The weights and thresholds of the resulting trained network were converted to integers by the quantization method shown in Fig. 5.

$$w_1x_1 + w_2x_2 + \cdots + w_nx_n - b \geq 0$$

$w_{max}$ $\qquad$ $w_{min}$

$$\frac{w_1}{w_{max} - w_{min}}x_1 + \frac{w_2}{w_{max} - w_{min}}x_2 + \cdots + \frac{w_n}{w_{max} - w_{min}}x_n - \frac{b}{w_{max} - w_{min}} \geq 0$$

$d$

$$Round(\frac{dw_1}{w_{max} - w_{min}})x_1 + Round(\frac{dw_2}{w_{max} - w_{min}})x_2 + \cdots + Round(\frac{dw_n}{w_{max} - w_{min}})x_n - Round(\frac{db}{w_{max} - w_{min}}) \geq 0$$

Figure 5: Quantization of weights and threshold.

Here $w_{max}$ and $w_{min}$ represent the maximum and the minimum values among $w_1 \cdots w_n$ and $b$, respectively. The constant $d$ was set to 200.

The quantized weights were used to construct a linear threshold network, the *derived linear threshold network*. From this network five probabilistic rule values ($P_1, \ldots, P_5$, to be shown in the following paragraphs) were extracted by the

Probabilistic Rule Extraction Algorithm, PE, as follows. Using the threshold functions (6) for the hidden nodes, the matrix $F$ was computed by means of the recursion (9) and the initialization (10); the value of the target is deduced from $F$ by equation (8). We compare these target values with the exact probability computed from the original Boolean functions and the ones obtained by the following simple sampling-based algorithm for extracting probabilistic rules directly from the trained sigmoid network.

**Algorithm 5** Probabilistic Rule Extraction by Sampling

$count = 0$
**for** $i = 1$ to $m$ **do**
  generate a random 0-1 assignment to the variables in the input layer;
  compute the 0-1 value of node $z$ output by the trained neural network;
  **if** $z = 1$ **then** $count++$;
**end for**
output $count/m$ as an approximation of $Pr(z = 1)$

For the 5-out-of-10 majority function the targets were $P_1 = Pr(z = 1|x_1 = 1)$, $P_2 = Pr(z = 1|x_1 = 1, x_2 = 1)$, $P_3 = Pr(z = 1|x_1 = 1, x_2 = 1, x_3 = 0)$, $P_4 = Pr(z = 1|x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0)$, and $P_5 = Pr(z = 1|x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1)$, and for each target the comparison value was obtained by running Algorithm 5 100 times with $m = 100$ samples.

The results, displayed in Table 1, show that the values output by our algorithm are essentially exact, whereas the results of Algorithm 5 depend on the samples that were generated. The small differences between the exact probabilities and those by PE are due to that the latter ones were computed using the linear threshold network obtained from the trained neural network.

| Algorithms | Parameters | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|---|
| exact probability | | 0.74609 | 0.85547 | 0.77344 | 0.65625 | 0.8125 |
| PE | | 0.7461 | 0.8555 | 0.7734 | 0.6563 | 0.8125 |
| PA | Mean | 0.7488 | 0.8601 | 0.7727 | 0.6588 | 0.8084 |
| | STD | 0.0451 | 0.0347 | 0.0423 | 0.0507 | 0.0314 |

Table 1: Results for the 5-out-of-10 majority function. PE is the probabilistic rule extraction algorithm. PA refers to Algorithm 5. Consult the text for the definitions of $P_1, \ldots, P_5$.

For the second Boolean function the targets were $P_1 = Pr(z = 1|x_1 = 1)$, $P_2 = Pr(z = 1|x_1 = 1, x_2 = 0)$, $P_3 = Pr(z = 1|x_1 = 1, x_2 = 0, x_3 = 1)$, $P_4 = Pr(z = 1|x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0)$, and $P_5 = Pr(z = 1|x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1)$. The results, displayed in Table 2, show that again the values output by our algorithm are essentially exact, and those of Algorithm 5 depend on the set of samples.

| Algorithms | Parameters | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|---|
| exact probability | | 0.76172 | 0.71094 | 0.8125 | 0.8125 | 0.8125 |
| PE | | 0.7617 | 0.7109 | 0.8125 | 0.8125 | 0.8125 |
| PA | Mean | 0.7631 | 0.7088 | 0.8129 | 0.8155 | 0.8121 |
| | STD | 0.0437 | 0.0401 | 0.0380 | 0.0373 | 0.0383 |

Table 2: Results for the conjunction of two 2-out-of-5 majority functions. PE is the probabilistic rule extraction algorithm. PA refers to Algorithm 5. Consult the text for the definitions of $P_1, \ldots, P_5$.

For the third Boolean function the targets were $P_1 = Pr(z = 1|x_1 = 0)$, $P_2 = Pr(z = 1|x_1 = 0, x_2 = 1)$, $P_3 = Pr(z = 1|x_1 = 0, x_2 = 1, x_3 = 0)$, $P_4 = Pr(z = 1|x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0)$, and $P_5 = Pr(z = 1|x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1)$. The results, displayed in Table 3, show that our algorithm gives more accurate values than PA gives.

| Algorithms | Parameters | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|---|
| exact probability | | 0.89258 | 0.95703 | 0.91406 | 0.82813 | 1 |
| PE | | 0.8828 | 0.9570 | 0.9141 | 0.8281 | 1 |
| PA | Mean | 0.8802 | 0.9602 | 0.9157 | 0.8249 | 1.0 |
| | STD | 0.0347 | 0.0196 | 0.0305 | 0.0385 | 0.0 |

Table 3: Results for $Maj_2(x_1, \ldots, x_5) \vee (x_6 \vee (x_7 \wedge (x_8 \vee (x_9 \wedge x_{10}))))$. PE is the probabilistic rule extraction algorithm. PA refers to Algorithm 5. Consult the text for the definitions of $P_1, \ldots, P_5$.

One advantage of our algorithm is that it is deterministic, so that its results do not depend on random samples. Another advantage is that it can extract probabilistic rules with very small probabilities.

In order to verify this property, we performed the following experiment. We designed a neural network with one hidden layer of 2 neurons and sigmoid activation functions for the Boolean function $x_1 \wedge \cdots \wedge x_{20}$ since it is hard to train a neural network for this function. From the designed network we constructed the derived threshold network with integer weights and threshold, and extracted from it $Pr(z = 1)$ as $9.53674 \times 10^{-7}$, which is in fact the exact value of that probability. In contrast, 100 trials of Algorithm 5 on the sigmoid network, each with 100 samples, were unable to find this non-zero probability, because the number of samples is too small. Possibly using a large number of samples will yield a non-zero probability, but it is difficult to estimate in advance the required number of samples.

Finally, we examine the possibility of extracting important attributes from trained neural networks using our probabilistic rule extraction algorithm. In this experiment, we train a neural network with $n$ binary inputs $x_i$ for $1 \leq i \leq n$ and one binary output $z$. We would like to compute $p(z = 1|x_i = 0)$ and $p(z = 1|x_i = 1)$ for all $x_i$ using our probabilistic rule extraction algorithm. We assume the trained neural network has one hidden layer (3 hidden neurons).

We tested our proposed algorithm on three different datasets.

1. The Votes Data

   This is the same dataset we considered in Sect. 5.1. The training was terminated once the loss was less than 0.02. The results in Fig. 6 show that physician-fee-freeze:$n$ is significantly more important than others. In fact, the accuracy of using only physician-fee-freeze:$n$ is 96.9%.

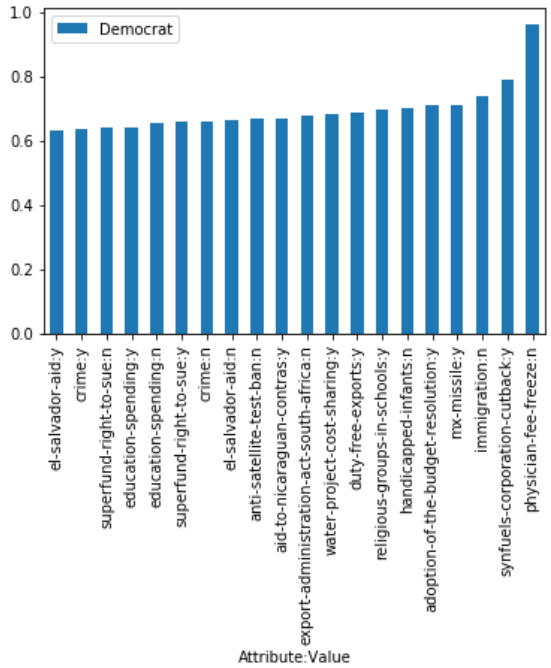


Figure 6: Rules with the Top 20 Highest Probabilities (Vote)

2. The Moral Reasoner Data

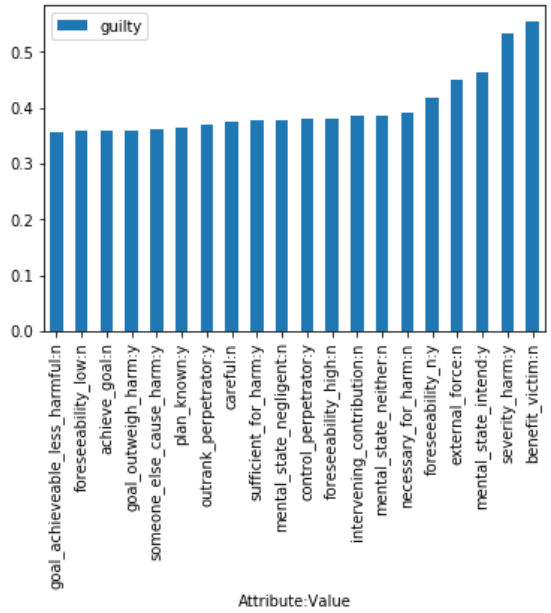


Figure 7: Rules with the Top 20 Highest Probabilities (Moral Reasoner)

This dataset, also part of the UCI Machine Learning Repository (Dua and Graff, 2017), contains 202 samples each having 23 attributes, of which 21 are binary. The other two discrete attributes are mental_state : $neither/negligent/reckless/intend$ and foreseeability : $n/low/high$. The classes are $guilty$ and $not\ guilty$.

We consider mental_state : $neither/negligent/reckless/intend$ as 4 binary attributes mental_state_neither : $n/y$, mental_state_negligent : $n/y$, mental_state_reckless : $n/y$ and mental_state_intend : $n/y$. Discrete attribute foreseeability : $n/low/high$ can be converted to 3 binary attributes in a similar way. Therefore there are 28 binary attributes. Training was halted once the training loss fell below 0.02.

It is fairly easy to construct from a few of the attributes that predict "guilty" with highest probabilities, as shown in Fig. 7, a rule with high accuracy. The following rule, for example, has an accuracy of 96%.

$$guilty : (benefit\_victim : n) \wedge (external\_force : n) \wedge (severity\_harm : y)$$

Interestingly, although the attribute $mental\_state\_intend : y$ has higher probability than $external\_force : n$, it does not participate in this rule.

The above results show that if the training loss is small, that is, the neural network approximates the corresponding dataset well, it is possible to obtain important rules through the probabilistic rule extraction algorithm. Next, we test whether it is possible to obtain some important rules when the training loss is not so small.

3. The Chess (King-Rook vs. King-Pawn) Data

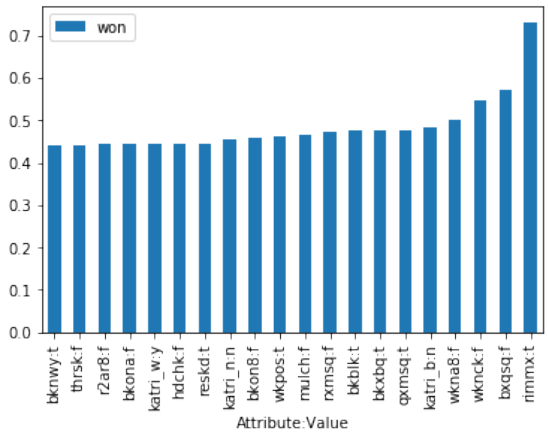


Figure 8: Rules with the Top 20 Highest Probabilities (Chess)

This dataset, also from the UCI Machine Learning Repository (Dua and Graff, 2017), includes 3196 samples. Each sample consists of the description of a legal initial board for this chess endgame and whether it is won by white, assuming optimal play by both sides. There are 36 discrete attributes of which 35 are binary. The discrete attribute $katri : b/n/w$ was converted into 3 binary attributes $katri\_b : n/y$, $katri\_n : n/y$, $katri\_w : n/y$. So there are 38 binary attributes. Classes are $won$ and $nowin$. The training process is terminated when the training loss is $0.02$. Fig. 8 shows rules with the top 20 highest probabilities. Among these rules, we found that the following Boolean rule achieved an accuracy of $90.4\%$.

$$won : (rimmx : t) \vee (wknck : f) \wedge (bxqsq : f)$$

## 6 Conclusions

In this paper we presented several theoretical results for extracting knowledge from trained neural networks.

We described first an algorithm for extracting a Boolean function from a trained neural network. The algorithm is efficient if the linear threshold functions in the neural network can be represented as nested canalyzing functions, $k$-out-of-$n$ majority functions, or some combinations of the two. If that is not the case, or if the neural network has hidden layers, the extracted Boolean function may turn out to be very large.

In order to cope with hidden layers, we limited the generation of rule terms to the third order. Computational experiments show that even if there are three hidden layers, the extracted Boolean functions approximate the corresponding neural networks well. Since banning high-order terms may not be appropriate in some applications, one important future direction of work is to obtain small size Boolean functions by combining our algorithm with simplification algorithms for Boolean functions such as (El-Bakry and Mastorakis, 2009). Although this modification may lead to generation of simple and understandable rules, the computational complexity in the first phase remains the same. Consequently, reduction of the computational complexity of the first phase is another important future work.

We also proposed an algorithm that extracts probabilistic relations between the input values and the output value in the form of conditional probabilities. Although this problem is NP-hard in general, our proposed algorithm works in pseudo-polynomial time if the number of neurons in hidden layers is bounded by a constant. Compared with a naive sampling-based algorithm, our algorithm has the advantages that it is deterministic, it outputs the exact probabilities if linear threshold functions are assigned to all nodes, and it does not miss rare relations. The potential usefulness of the algorithm was demonstrated by means of computational experiments. One disadvantage of this algorithm is that it needs to perform an exhaustive search for the second layer (i.e., the first hidden layer). Therefore, this algorithm is not practical if there are many neurons in the second layer and thus some improvements should be done. Use of sparse modeling (Rish and Grabarnik, 2014) might be useful to find important input attributes without exhaustive search. Therefore, incorporation of sparse modeling into our proposed methods is important future work. Another problem is that we assume that the input neurons obey independent distributions. Although computational experiments show the effectiveness of this algorithm under the independence assumption, dependencies between input attributes is almost inevitable in real world data. It might be useful to study the quantitative relation between the difference and the degree of dependencies. Thus we leave it as an open problem.

## References


M. Anthony. *Discrete mathematics of neural networks: selected topics*, volume 8. SIAM, Philadelphia, PA, USA, 2001.

M. Anthony. Decision lists and related classes of boolean functions. In *Crama, Yves and Hammer, Peter L., Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 577–598, New York, NY, USA, 2010. Cambridge University Press. ISBN 0521847524, 9780521847520.

M. G. Augusta and T. Kathirvalavakumar. Reverse engineering the neural networks for rule extraction in classification problems. *Neural processing letters*, 35(2):131–150, 2012.

P. Barth. Linear 0–1 inequalities and extended clauses. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 40–51. Springer, 1993.

D. Dua and C. Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

H. M. El-Bakry and N. Mastorakis. A fast computerized method for automatic simplification of boolean functions. In *WSEAS International Conference. Proceedings. Mathematics and Computers in Science and Engineering*, number 9. WSEAS, 2009.

A. Fauré, A. Naldi, C. Chaouiya, and D. Thieffry. Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle. *Bioinformatics*, 22(14):e124–e131, 2006.

M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness (series of books in the mathematical sciences), ed. *Computers and Intractability*, page 340, 1979.

S. E. Harris, B. K. Sawhill, A. Wuensche, and S. Kauffman. A model of transcriptional regulatory networks based on biases in the observed regulation rules. *Complexity*, 7:23–40, 2002.

M. Ishikawa. Rule extraction by successive regularization. *Neural Networks*, 13(10):1171–1183, 2000.

A. S. Jarrah, B. Raposa, and R. Laubenbacher. Nested canalyzing, unate cascade, and polynomial functions. *Physica D: Nonlinear Phenomena*, 233(2):167–174, 2007.

Q. V. Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

M. A. Mazurowski, P. A. Habas, J. M. Zurada, J. Y. Lo, J. A. Baker, and G. D. Tourassi. Training neural network classifiers for medical decision making: The effects of imbalanced datasets on classification performance. *Neural networks*, 21(2-3):427–436, 2008.

A. A. Melkman, X. Cheng, W.-K. Ching, and T. Akutsu. Identifying a probabilistic boolean threshold network from samples. *IEEE transactions on neural networks and learning systems*, 29(4):869–881, 2018.

R. O'Donnell. *Analysis of boolean functions*. Cambridge University Press, New York, NY, USA, 2014.

A. Paszke, S. Gross, S. Chintala, and G. Chanan. PyTorch. https://pytorch.org/get-started/locally/, 2017. Accessed: 2018-12-17.

F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer, New York, NY, USA, 1985.

J. R. Quinlan. *C4. 5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

M. T. Ribeiro, S. Singh, and C. Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM, 2016.

I. Rish and G. Grabarnik. *Sparse modeling: theory, algorithms, and applications*. CRC press, 2014.

E. W. Saad and D. C. Wunsch II. Neural network explanation using inversion. *Neural Networks*, 20(1):78–93, 2007.

M. I. Shamos. *Computational geometry.* Ph. D. thesis, Yale University, 1978.

D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.

J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

H. Tsukimoto. Extracting rules from trained neural networks. *IEEE Transactions on Neural Networks*, 11(2):377–389, 2000.

L. Zhao and C. E. Thorpe. Stereo-and neural network-based pedestrian detection. *IEEE Transactions on intelligent transportation systems*, 1(3):148–154, 2000.