Extracting Reduced Logic Programs from Artificial Neural Networks

Jens Lehmann[1], Sebastian Bader[2], Pascal Hitzler[3]
[1]Department of Computer Science, Universität Leipzig, Germany
[2]International Center for Computational Logic,
Technische Universität Dresden, Germany
[3]AIFB, Universität Karlsruhe (TH), Germany

Abstract

Artificial neural networks can be trained to perform excellently in many application areas. Whilst they can learn from raw data to solve sophisticated recognition and analysis problems, the acquired knowledge remains hidden within the network architecture and is not readily accessible for analysis or further use: Trained networks are *black boxes*. Recent research efforts therefore investigate the possibility to extract symbolic knowledge from trained networks, in order to analyze, validate, and reuse the structural insights gained implicitly during the training process. In this paper, we will study how knowledge in form of propositional logic programs can be obtained in such a way that the programs are as *simple* as possible — where *simple* is being understood in some clearly defined and meaningful way.

1 Introduction and Motivation

The success of the neural networks machine learning technology for academic and industrial use is undeniable. There are countless real uses spanning over many application areas such as image analysis, speech and pattern recognition, investment analysis, engine monitoring, fault diagnosis, etc. During a training process from raw data, artificial neural networks acquire expert knowledge about the problem domain, and have the ability to generalize this knowledge to similar but previously unencountered situations in a way which often surpasses the abilities of human experts.

The knowledge obtained during the training process, however, is hidden within the acquired network architecture and connection weights, and not directly accessible for analysis, reuse, or improvement, thus limiting the range of applicability of the neural networks technology. For these purposes, the knowledge would be required to be available in structured symbolic form, most preferably expressed using some logical framework.

Suitable methods for the extraction of knowledge from neural networks are therefore being sought within many ongoing research projects worldwide, see [1, 2, 14, 27, 30, 33, 37] to mention a few recent publications. One of the prominent approaches seeks to extract knowledge in the form of logic programs, i.e. by describing the input-output behavior of a network in terms of material implication or *rules*. More precisely, activation ranges of input and output nodes are identified with truth values for propositional variables, leading directly to the description of the input-output behavior of the network in terms of a set of logic program rules.

This naive approach is fundamental to the rule extraction task. However, the set of rules thus obtained is usually highly redundant and typically turns out to be as hard to understand as the trained network itself. One of the main issues in propositional rule extraction is therefore to alter the naive approach in order to obtain a *simpler* set of rules, i.e. one which appears to be more meaningful and intelligible.

Within the context of our own broader research efforts described e.g. in [3, 5, 6, 7, 8, 9, 4, 22, 24], we seek to understand rule extraction within a learning cycle of (1) initializing an untrained network with background knowledge, (2) training of the network taking background knowledge into account, (3) extraction of knowledge from the trained network, see Figure 1, as described for example in [7, 8, 17]. While our research efforts mainly concern first-order neural-symbolic integration, we consider the propositional case to be fundamental for our studies.

We were surprised, however, that the following basic question apparently had

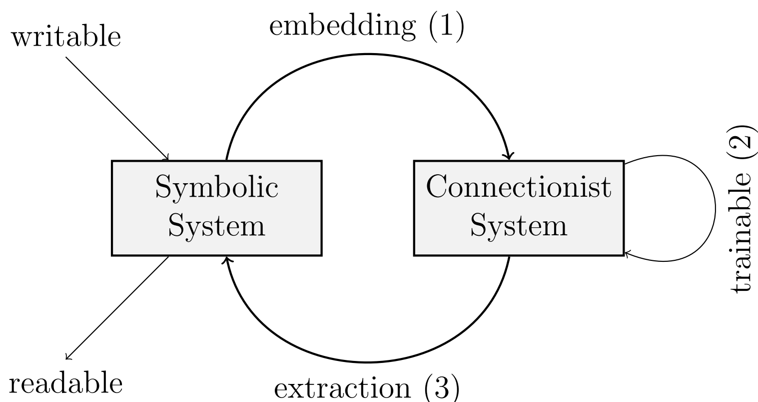

Figure 1: The Neural-Symbolic Learning Cycle

not been answered yet within the available literature: *Using the data obtained from the naive rule extraction approach described above — when is it possible to obtain a unique irredundant representation of the extracted data?* While we believe that applicable extraction methods will have to deviate from the exact approach implicitly assumed in the question, we consider an answer important for providing a fundamental understanding of the issue. This paper is meant to settle the question to a satisfactory extent.

More precisely, we prove formally that a unique irredundant representation can be obtained if the use of negation within the knowledge base is forbidden, i.e. when considering *definite* logic programs — and we also clarify formally what we mean by *redundancy* in this case. We also show that in the presence of negation, i.e. for *normal* logic programs, unique representations cannot be obtained in general. However, we present heuristic algorithms which optimize naive approaches by retrieving programs with less redundancies, and we report on experimental evaluations which show the usefulness of these algorithms. To the best of our knowledge, this paper constitutes the most comprehensive treatment of the subject matter which has been done so far. We would like to stress, however, that we consider our results to be of fundamental nature, i.e. we do not claim that they can be of direct practical use in the form presented herein.

The structure of the paper is as follows. After some preliminaries reviewed in Sections 2 and 3, we will present our main theoretical result on the extraction of a unique irredundant definite logic program in Section 4, together with a corresponding algorithm. How to remove redundancies in normal logic programs is discussed in Section 5, while a more powerful extraction algorithms for these

programs is presented in Section 6. In Section 7 we show formally that the main theoretical result from Section 4 cannot be carried over to normal programs. An experimental evaluation of our algorithms is presented in Section 8. A discussion of related work follows in Section 9 and some final conclusions are drawn in Section 10.

Acknowledgments

Jens Lehmann is supported by the German Federal Ministry of Education and Research (BMBF) under the SoftWiki project 01ISF02B. Sebastian Bader was supported by the GK334 of the German Research Foundation (DFG). Pascal Hitzler is supported by the Deutsche Forschungsgemeinschaft (DFG) under the ReaSem project.

2 Logic Programs

We first introduce some standard notation for logic programs, roughly following [31].

A predicate in propositional logic is also called an *atom*. A *literal* is an atom or a negated atom. A *(Horn) clause* in propositional logic is of the form $q \leftarrow l_1, \ldots, l_n$ with $n \geq 0$, where q is an atom and all l_i with $1 \leq i \leq n$ are literals, and q is called the *head* and $l_1, \ldots, l_n$ the *body* of the clause. Clause bodies are understood to be conjunctions. If all l_i are atoms a clause is called *definite*. The number of literals in the body of a clause is called the *length* of the clause. A *(normal propositional) logic program* is a finite set of clauses, a *definite (propositional) logic program* is a finite set of definite clauses.

An *interpretation* maps predicates to $true$ or $false$. We will usually identify an interpretation with the set of predicates which it maps to $true$. An interpretation is extended to literals, clauses and programs in the usual way. A *model* of a clause C is an interpretation I which maps C to $true$ (in symbols: $I \models C$). A model of a program $\mathcal{P}$ is an interpretation which maps every clause in $\mathcal{P}$ to $true$.

Given a logic program $\mathcal{P}$, we denote the (finite) set of all atoms occurring in it by $B_{\mathcal{P}}$, and the set of all interpretations of $\mathcal{P}$ by $I_{\mathcal{P}}$; note that $I_{\mathcal{P}}$ is the powerset of the (finite) set $B_{\mathcal{P}}$ of all atoms occurring in $\mathcal{P}$.

As a neural network can be understood as a function between its input and output layer, we require a similar perspective on logic programs. This is provided

by the standard notion of a semantic operator, which is used to describe the meaning of a program in terms of operator properties [31]. We will elaborate on the relation to neural networks in Section 3. The *immediate consequence operator* $T_{\mathcal{P}}$ associated with a given logic program $\mathcal{P}$ is defined as follows:

Definition 2.1. *$T_{\mathcal{P}}$ is a mapping from interpretations to interpretations defined in the following way for an interpretation I and a program $\mathcal{P}$:*

$$T_{\mathcal{P}}(I) := \{q \mid q \leftarrow \mathrm{B} \text{ is a clause in } \mathcal{P} \text{ and } I \models \mathrm{B}\}.$$

If the underlying program is definite we will call $T_{\mathcal{P}}$ *definite*. An important property of definite $T_{\mathcal{P}}$-operators is monotonicity, i.e. $I \subseteq J$ implies $T_{\mathcal{P}}(I) \subseteq T_{\mathcal{P}}(J)$. The operators $T_{\mathcal{P}}$ for a program $\mathcal{P}$ and $T_{\mathcal{Q}}$ for a program $\mathcal{Q}$ are *equal* if they are pointwise equal, i.e. if we have $T_{\mathcal{P}}(I) = T_{\mathcal{Q}}(I)$ for all interpretations I. In this case, we call the programs $\mathcal{P}$ and $\mathcal{Q}$ *equivalent*.

As mentioned in the introduction, we are interested in extracting *small* programs from networks. We will use the obvious measure of *size* of a program $\mathcal{P}$, which is defined as the sum of the number of all (not necessarily distinct) literals in all clauses in $\mathcal{P}$. A program $\mathcal{P}$ is called (strictly) *smaller* than a program $\mathcal{Q}$, if its size is (strictly) less than the size of $\mathcal{Q}$.

As already noted, the immediate consequence operator will serve as a link between programs and networks, i.e. we will be interested in logic programs *up to equivalence*. Consequently, a program will be called *minimal*, if there is no strictly smaller equivalent program.

The notion of minimality just introduced is difficult to operationalize. We thus introduce the notion of *reduced* program; the relationship between reduction and minimality will become clear later on in Corollary 4.4. Reduction is described in terms of *subsumption*, which conveys the idea of redundancy of a certain clause C_2 in presence of another clause C_1. If in a given program $\mathcal{P}$, we have that C_1 subsumes C_2, we find that the $T_{\mathcal{P}}$-operator of the program does not change after removing C_2.

Definition 2.2. *A clause is said to be* consistent *iff the body does not contain a predicate and its negation.*

Definition 2.3. *A clause $C_1 : h \leftarrow p_1, \ldots, p_a, \neg q_1, \ldots, \neg q_b$ is said to* subsume *the clause $C_2 : h \leftarrow r_1, \ldots, r_c, \neg s_1, \ldots, \neg s_d$, iff we have $\{p_1, \ldots, p_a\} \subseteq \{r_1, \ldots, r_c\}$ and $\{q_1, \ldots, q_b\} \subseteq \{s_1, \ldots, s_d\}$.*

Algorithm 1: Constructing a reduced logic program

Input: An arbitrary program $\mathcal{P}$.
Output: A reduced logic program $\mathcal{Q}$ with $T_\mathcal{P} = T_\mathcal{Q}$.

1 Initialize $\mathcal{Q} = \mathcal{P}$.
2 **while** *one of the following reductions applies* **do**
3   **if** *there is an inconsistent clause C in $\mathcal{Q}$* **then**
4   Remove C.
5   **if** *there is a clause in $\mathcal{Q}$ whose body contains the literal L twice* **then**
6   Remove one occurrence of L from the body.
7   **if** *there are $C_1 \neq C_2$ in $\mathcal{Q}$ such that C_1 subsumes C_2* **then**
8   Remove C_2.
9 Return $\mathcal{Q}$ as result

Definition 2.4. *A program $\mathcal{P}$ is called* reduced *if the following properties hold:*

1. *Every clause in $\mathcal{P}$ is consistent.*
2. *No literal appears more than once in some clause body.*
3. *There are no clauses $C_1 \neq C_2$ in $\mathcal{P}$, such that C_1 subsumes C_2.*

Humans usually write reduced logic programs. Using Definition 2.4, we can define the naive Algorithm 1 for reducing logic programs: Simply check every condition separately on every clause, and remove the subsumed, respectively irrelevant, symbols or clauses. Performing steps of this algorithm is called *reducing* a program.

Proposition 2.5. *If Q is a reduced version of the propositional logic program $\mathcal{P}$, then $T_\mathcal{P} = T_\mathcal{Q}$.*

Proof sketch. Algorithm 1 will change the given program in three cases. (1) Inconsistent clauses are removed. This does not change the associated operator, because an inconsistent clause does not contribute to it. (2) Double occurrences of literals are removed, which does not change the behavior of the clause. (3) Subsumed clauses are removed. But whenever a clause C_2 is subsumed by C_1, we find that under all interpretations that satisfy the body of C_1 that the body of C_2 is also mapped to true. Therefore, the $T_\mathcal{P}$-operator does not change while removing C_2 from $\mathcal{P}$. □

3 Neural-Symbolic Integration

For the purpose of this paper, an *artificial neural network*, also called *connectionist system*, consists of (a finite set of) *nodes* or *units* and weighted directed connections between them. The weights are understood to be real numbers. The network updates itself in discrete time steps. At every point in time, each unit carries a real-numbered *activation*. The activation is computed based on the current input of the unit from the incoming weighted connections from the previous time step, as follows. Let $v_1, \ldots, v_n$ be the activation of the predecessor units for a unit k at time step t, and let $w_1, \ldots, w_n$ be the weights of the connections between those units and unit k, then the *input* of unit k is computed as $i_k = \sum_i w_i \cdot v_i$. The activation of the unit at time step $t + 1$ is obtained by applying a simple function to its input, e.g. a threshold or a sigmoidal function.

We would like to remark, that the mathematical properties which we will show in the following, are independent of any concrete neural network paradigm. They may even prove useful for the understanding of biological neural networks, though this remains to be investigated, and the authors' interests lie primarily in understanding artificial connectionist systems. We refer to [11] for general background on artificial neural networks.

More specifically, we consider so-called 3-layer feed forward networks with threshold activation functions, as depicted in Figure 2. The nodes in the leftmost layer are called the *input nodes* and the nodes in the rightmost layer are called the *output nodes* of the network. A network can be understood as computing the function determined by propagating some input activation to the output layer.

In order to connect the input-output behavior of a neural network with the immediate consequence operator of a logic program, we interpret the input and output nodes to be propositional variables. Activations above a certain threshold are interpreted as *true*, others as *false*. In [22, 25], an algorithm was presented for constructing a neural network for a given $T_{\mathcal{P}}$-operator, thus providing the initialization step depicted in Figure 1. Without going into the details, we will give the basic principles here. For each atom in the program there is one unit in the input and output layer of the network, and for each clause there is a unit in the hidden layer. The connections between the layers are set up such that the input-output behavior of the network matches the $T_{\mathcal{P}}$-operator. The basic idea is depicted in Figure 2, and an example-run of the network is shown in Figure 3. The algorithm was generalized to sigmoidal activation functions in [17], thus enabling the use of powerful learning algorithms based on backpropagation [11]. The resulting *Connectionist Inductive Learning and Logic Programming System* (CILP), also

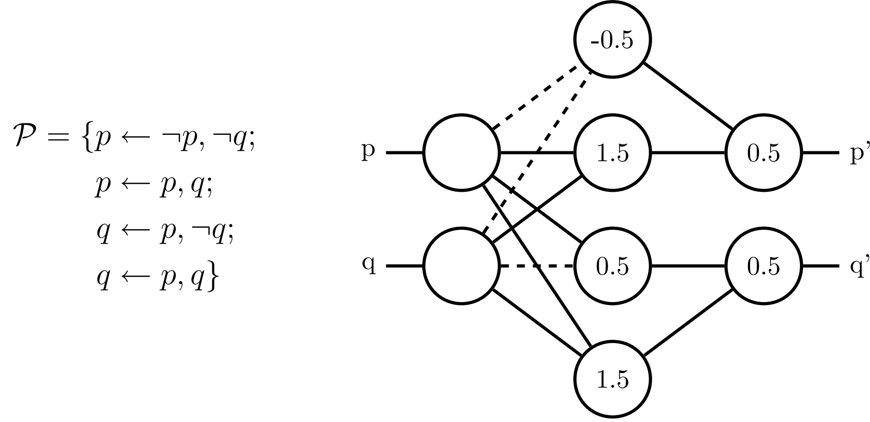

Figure 2: The 3-layer network constructed to implement the $T_{\mathcal{P}}$-operator of the given program $\mathcal{P}$. Connections with weight 1 are depicted solid, those with weight -1 are dashed. The numbers denote the thresholds of the units.

comprises extraction capabilities described in [14], and we will return to this in Section 9.

The representation of the $T_{\mathcal{P}}$-operator as a feedforward network is the base of the so-called *core method* [7] for neural-symbolic integration. In this paper, however, we are concerned with the extraction of logic programs from neural networks. The naive, sometimes called *global* or *pedagogical* approach is to activate the input layer of the given network with all possible interpretations, and to read off the corresponding interpretations in the output layer. We thus obtain a mapping $f : I_{\mathcal{P}} \rightarrow I_{\mathcal{P}}$ as *target function* for the knowledge extraction by interpreting it as an immediate consequence operator. The task which remains is to find a logic program $\mathcal{P}$ such that $T_{\mathcal{P}} = f$, and furthermore, to do this such that $\mathcal{P}$ is as simple as possible.

We start with a naive extraction by *"Full Exploration"*, detailed in Algorithms 2 and 3, for definite and normal logic programs, respectively. We will find that the extraction of definite programs is easier and theoretically more satisfactory. However, negation is perceived as highly desirable because it allows to express knowledge more naturally. We give an example for full exploration in the normal case. Algorithm 3 allows to state the following proposition.

Proposition 3.1. *For every mapping $f : I_{\mathcal{P}} \rightarrow I_{\mathcal{P}}$, we can construct a propositional logic program $\mathcal{P}$ with $T_{\mathcal{P}} = f$.*

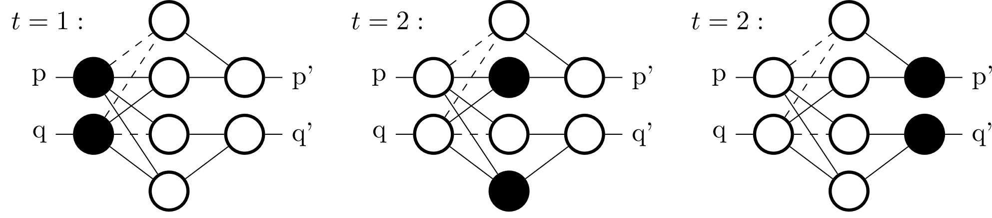

Figure 3: A run of the network depicted in Figure 2 for the interpretation $I = \{p, q\}$. A unit is depicted in black, if its activation is 1. At time $t = 0$ the corresponding units in the input layer are activated to 1. This activation is propagated to the hidden layer and results in two active units there. Finally, it reaches the output layer, i.e. $T_{\mathcal{P}}(I) = \{p, q\}$.

Algorithm 2: Full Exploration — Definite

Input: A monotone mapping $f : I_{\mathcal{P}} \rightarrow I_{\mathcal{P}}$.
Output: A definite logic program $\mathcal{P}$ with $T_{\mathcal{P}} = f$.

1 Initialize $\mathcal{P} = \emptyset$.
2 **foreach** *interpretation* $I := \{r_1, \ldots, r_a\} \in I_{\mathcal{P}}$ **do**
3   **foreach** $h \in f(I)$ **do**
4   Add the clause $h \leftarrow r_1, \ldots, r_a$ to $\mathcal{P}$.
5 Return $\mathcal{P}$ as result

Example 3.2. *Let $B_{\mathcal{P}} = \{p, q\}$ and the mapping f be obtained by querying the network depicted in Figure 2. Using Algorithm 3, we re-obtain program $\mathcal{P}$ as given in Figure 2, and find that $T_{\mathcal{P}} = f$ holds.*

$$
f = \{ \quad \emptyset \mapsto \{p\} \qquad \qquad \mathcal{P} = \{p \leftarrow \neg p, \neg q;
$$
$$
\{p\} \mapsto \{q\} \qquad \qquad p \leftarrow p, q;
$$
$$
\{q\} \mapsto \emptyset \qquad \qquad q \leftarrow p, \neg q;
$$
$$
\{p, q\} \mapsto \{p, q\}\} \qquad \qquad q \leftarrow p, q\}
$$

Note that programs obtained using Algorithms 2 or 3 are in general neither reduced nor minimal. In order to obtain simpler programs, there are basically two possibilities. On the one hand we can extract a large program using e.g. Algorithms 2 or 3 and refine it. This general idea was first described in [25], but not spelled out using an algorithm. On the other hand, we can build a program from scratch. Both possibilities will be pursued in the sequel.

Algorithm 3: Full Exploration — Normal

Input: An arbitrary mapping $f : I_{\mathcal{P}} \to I_{\mathcal{P}}$.
Output: A normal logic program $\mathcal{P}$ with $T_{\mathcal{P}} = f$.

1 Initialize $\mathcal{P} = \emptyset$.
2 **foreach** *interpretation* $I := \{r_1, \ldots, r_a\} \in I_{\mathcal{P}}$ **do**
3   Let $B_{\mathcal{P}} \setminus I = \{s_1, \ldots, s_b\}$.
4   **foreach** $h \in f(I)$ **do**
5   Add the clause $h \leftarrow r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ to $\mathcal{P}$.
6 Return $\mathcal{P}$ as result

4 Extracting Reduced Definite Programs

First, we will discuss the simpler case of definite logic programs. We will derive an algorithm which returns only minimal programs, and we will show that the notion of minimal program coincides with that of a reduced program, thus serving both intuitions at the same time. To obtain a minimal program, we will employ a linear order $\prec$ on the space of interpretations which has the property that $I \prec J$ whenever $I \subseteq J$. Since all considered interpretations are finite, such a linear order trivially exists. We then proceed as in Algorithm 2. But by querying the interpretations according to the order, we will add only necessary clauses and obtain a minimal program. Algorithm 4 shows the extraction of a reduced definite program from a monotone mapping f. The correctness and minimality of the result is established in Propositions 4.1 and 4.2 respectively.

Proposition 4.1. *Let $T_{\mathcal{P}}$ be a definite consequence operator and $\mathcal{Q}$ be the result of Algorithm 4, obtained for $f = T_{\mathcal{P}}$. Then $T_{\mathcal{P}} = T_{\mathcal{Q}}$.*

Proof. We will show that $T_{\mathcal{P}}(I) = T_{\mathcal{Q}}(I)$ for an arbitrary $I = \{p_1, \ldots, p_n\}$ by showing $T_{\mathcal{P}}(I) \subseteq T_{\mathcal{Q}}(I)$ and $T_{\mathcal{P}}(I) \supseteq T_{\mathcal{Q}}(I)$.

To show $T_{\mathcal{P}}(I) \subseteq T_{\mathcal{Q}}(I)$ we assume $q \in T_{\mathcal{P}}(I)$ and show that $q \in T_{\mathcal{Q}}(I)$ follows: We know that the algorithm will treat I and q (because for every interpretation I every element in $T_{\mathcal{P}}(I)$ is investigated). Then we have to distinguish two cases.

1. There already exists a clause $q \leftarrow q_1, \ldots, q_m$ with $\{q_1, \ldots, q_m\} \subseteq I$ in $\mathcal{Q}$. Then by definition $q \in T_{\mathcal{Q}}(I)$.

Algorithm 4: Extracting a Reduced Definite Program

Input: A monotone mapping $f : I_{\mathcal{P}} \to I_{\mathcal{P}}$.
Output: A definite logic program $\mathcal{P}$ with $T_{\mathcal{P}} = f$.

1 Fix an order $\prec$ on $I_{\mathcal{P}}$ with $I \prec J$ if $|I| < |J|$.
2 Initialize $\mathcal{Q} = \emptyset$.
3 **foreach** $I := \{p_1, \ldots, p_n\} \in I_{\mathcal{P}}$ *(ascending according to* $\prec$*)* **do**
4   **foreach** $q \in f(I)$ **do**
5   **if** *there is no* $(q \leftarrow q_1, \ldots, q_m) \in \mathcal{Q}$ *with* $\{q_1, \ldots, q_m\} \subseteq I$ **then**
6   add the clause $q \leftarrow p_1, \ldots, p_n$ to $\mathcal{Q}$.

7 Return $\mathcal{Q}$ as the result.

2. If there is no such clause $q \leftarrow p_1, \ldots, p_n$ yet, it is added to $\mathcal{Q}$, hence we have $q \in T_{\mathcal{Q}}(I)$.

Conversely, we show $T_{\mathcal{P}}(I) \supseteq T_{\mathcal{Q}}(I)$ by assuming $q \in T_{\mathcal{Q}}(I)$ and deriving $q \in T_{\mathcal{P}}(I)$: If $q \in T_{\mathcal{Q}}(I)$ we have by definition of $T_{\mathcal{Q}}$ a clause $q \leftarrow q_1, \ldots, q_m$ with $\{q_1, \ldots, q_m\} \subseteq I$. This means that the extraction algorithm must have treated the case $q \in T_{\mathcal{P}}(J)$ with $J = \{q_1, \ldots, q_m\}$. Since $T_{\mathcal{P}}$ is monotonic (it is the operator of a definite program) and $J \subseteq I$ we have $T_{\mathcal{P}}(J) \subseteq T_{\mathcal{P}}(I)$, hence q is also an element of $T_{\mathcal{P}}(I)$. □

Proposition 4.2. *The output of Algorithm 4 is a reduced definite propositional logic program.*

Proof. Obviously the output of the algorithm is a definite program $\mathcal{Q}$, because it generates only definite clauses. We have to show that the resulting program is reduced. For a proof by contradiction we assume that $\mathcal{Q}$ is not reduced. According to Definition 2.4, there are three possible reasons for this: (1) The program contains an inconsistent clause. (2) A predicate symbol appears more than once in the body of a clause. (3) There are two different clauses C_1 and C_2 in $\mathcal{Q}$, such that C_1 subsumes C_2.

Case (1) and (2) are impossible, because neither inconsistent clauses are constructed, nor literals are added twice. To show that no subsumed clause is constructed (3), we let C_1 be $h \leftarrow p_1, \ldots, p_a$ and C_2 be $h \leftarrow q_1, \ldots, q_b$ and assume $\{p_1, \ldots, p_a\} \subseteq \{q_1, \ldots, q_b\}$. As abbreviations we use $I = \{p_1, \ldots, p_a\}$ and $J = \{q_1, \ldots, q_b\}$. Because of (2) being impossible, we know $|I| = a$ and $|J| = b$

and $|I| < |J|$. This means the algorithm has treated I (and $h \in f(I)$) before J (and $h \in f(J)$). C_1 was generated by treating I and h, because C_1 exists and can only be generated through I and h. While treating J and h, the algorithm checks for clauses $h \leftarrow r_1, \ldots, r_m$ with $\{r_1, \ldots, r_m\} \subseteq J$. Because C_1 is such a clause, C_2 cannot be a clause in $\mathcal{Q}$, which is a contradiction and completes the proof. □

Propositions 4.1 and 4.2 show that the output of the extraction algorithm is in fact a reduced definite program, which has the desired operator. Please note that we require the input of the algorithm to be an operator of a definite program, i.e. to be a monotonic mapping $f : I_{\mathcal{P}} \rightarrow I_{\mathcal{P}}$.

There are interesting points regarding the efficiency of the presented algorithm. Instead of storing all interpretations it is easy to write a successor function, which returns the "next" interpretation to be handled. Thus it is not necessary to actually store interpretations and therefore the space complexity is very low, indeed only Q and the current interpretation need to be stored. The bottleneck is the time complexity, which is exponential with respect to the number of predicates. However, it is also exponential with respect to the maximum length of a clause in Q, because for an input $|I| = n$ the algorithm generates Horn clauses of length n only. Thus if we know a limit n of the number of elements in a body of a Horn clause in advance, we can reduce time complexity and maintain the properties proved above, by stopping the algorithm if $|I| > n$.

We proceed to show that the reduced program obtained by the algorithm is unique. The following theorem together with Corollary 4.4 are two of the main theoretical results in this paper.

Theorem 4.3. *For any operator $T_{\mathcal{P}}$ of a definite propositional logic program $\mathcal{P}$ there is exactly one reduced definite propositional logic program $\mathcal{Q}$ with $T_{\mathcal{P}} = T_{\mathcal{Q}}$.*

Proof. Assume we have an operator $T_{\mathcal{P}}$ of a definite program $\mathcal{P}$. With Algorithm 4 applied to $f = T_{\mathcal{P}}$ and Propositions 4.1 and 4.2 it follows that there is a reduced definite program $\mathcal{Q}$ with $T_{\mathcal{P}} = T_{\mathcal{Q}}$. We have to show that there cannot be more than one program with this property.

To prove this, we assume (by contradiction) that we have two different reduced definite programs P_1 and P_2 with $T_{\mathcal{P}} = T_{P_1} = T_{P_2}$. Two programs being different means that there is at least one clause existing in one of the programs which does not exist in the other program, say a clause C_1 in P_1 which is not in P_2. C_1 is some definite clause of the form $h \leftarrow p_1, \ldots, p_m$. By definition of $T_{\mathcal{P}}$, we have $h \in T_{P_1}(\{p_1, \ldots, p_m\})$. Because T_{P_1} and T_{P_2} are equal we also have $h \in T_{P_2}(\{p_1, \ldots, p_m\})$. This means that there is a clause C_2 of the form

$h \leftarrow q_1, \ldots, q_n$ with $\{q_1, \ldots, q_n\} \subseteq \{p_1, \ldots, p_n\}$ in P_2. Applying the definition of $T_\mathcal{P}$ again, this means that $h \in T_{P_2}(\{q_1, \ldots, q_n\})$ and $h \in T_{P_1}(\{q_1, \ldots, q_n\})$. Thus, we know that there must be a clause C_3 of the form $h \leftarrow r_1, \ldots, r_o$ with $\{r_1, \ldots, r_o\} \subseteq \{q_1, \ldots, q_n\}$ in P_1.

C_3 subsumes C_1, because $\{r_1, \ldots, r_o\} \subseteq \{q_1, \ldots, q_n\} \subseteq \{p_1, \ldots, p_m\}$ and they have the same head. We know, that by our assumption C_1 is not equal to C_2, because C_1 is not equal to any clause in P_2. Additionally, we know that $|\{p_1, \ldots, q_m\}| = m$ and $|\{q_1, \ldots, q_n\}| = n$, because P_1 and P_2 are reduced, i.e. no predicate appears more than once in any clause body. So we have $\{q_1, \ldots, q_n\} \subset \{p_1, \ldots, p_m\}$. Because C_3 has at most as many elements in its body as C_2, we know that C_1 is not equal to C_3. That means that P_1 contains two different clauses C_1 and C_3, where C_3 subsumes C_1. This contradicts P_1 being reduced. □

This shows that each algorithm extracting reduced definite programs from a neural network must return the same result as Algorithm 4. We can now also obtain that the notion of reduced program coincides with that of minimal program, which shows that Algorithm 4 also extracts the *least* program in terms of size.

Corollary 4.4. *If $\mathcal{P}$ is a reduced definite propositional logic program, then it is least in terms of size.*

Proof. Let $\mathcal{Q}$ be any program with $T_\mathcal{Q} = T_\mathcal{P}$. If $\mathcal{Q}$ is reduced, then it must be equal to $\mathcal{P}$ by Theorem 4.3. Assuming $\mathcal{Q}$ is not reduced, we find that the reduced program $\mathcal{Q}_{red}$ is definite, by Definition 2.4 smaller than $\mathcal{Q}$, and $T_\mathcal{P} = T_\mathcal{Q}$. From Theorem 4.3 we know that there is only one reduced definite program with operator $T_\mathcal{P}$, so we have $\mathcal{P} = \mathcal{Q}_{red}$. Because $\mathcal{Q}_{red}$ is smaller than $\mathcal{Q}$, we find that $\mathcal{P}$ is also smaller than $\mathcal{Q}$. Hence, $\mathcal{P}$ is smaller than any non-reduced program. □

The results just given, show that there is a unique desired choice for the result of the extraction in case on a monotonic input-output mapping of the network. Note, that this certainly does not answer or even address how to obtain a meaningful interpretation of the extracted program, a task which is important but out of scope for the more fundamental issues addressed in this paper.

5 Reducing Normal Logic Programs

As discussed in Section 3, it is possible to extract a normal logic program $\mathcal{P}$ from a neural network, such that the behavior of the associated $T_\mathcal{P}$-operator and the input-output-mapping of the network are identical. But the program obtained from the

naive Algorithm 3 in general yields an unwieldy program. In this section, we will show how to refine this logic program.

The first question to be asked is: Will we be able to obtain a result as strong as Theorem 4.3? The following example indicates a negative answer, and a formal assessment of the situation will follow later on in Proposition 7.1.

Example 5.1. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be defined as follows:*

$$\mathcal{P}_1 = \{p \leftarrow q; \quad p \leftarrow \neg q\} \qquad \mathcal{P}_2 = \{p \leftarrow\}$$

Obviously, in program $\mathcal{P}_1$, p does not depend on q. Hence, the two programs are equivalent but $\mathcal{P}_2$ is smaller than $\mathcal{P}_1$. We note, however, that $\mathcal{P}_2$ cannot be obtained from $\mathcal{P}_1$ by reduction in the sense of Definition 2.4.

Example 5.1 shows that the notion of reduction in terms of Definition 2.4 is insufficient for normal logic programs, whereas *size* obviously is a meaningful notion. A naive algorithm for obtaining minimal normal programs is easily constructed: As $B_{\mathcal{P}}$ is finite, so is the set of all possible normal programs over $B_{\mathcal{P}}$ (assuming we avoid multiple occurrences of atoms in the same clause body and multiple occurrences of the same clause). We can now search this set and extract from it all programs whose immediate consequence operator coincides with the target function, and subsequently we can extract all minimal programs by doing a complete search. This algorithm is obviously too naive to be practical.

For the moment, we will shortly discuss possibilities for refining the set obtained by Algorithm 3 (Full Exploration). We start with two examples.

Example 5.2. *Let $\mathcal{P}_1$ (as in Example 3.2) and $\mathcal{P}_2$ be defined as follows:*

$$\mathcal{P}_1 = \{ p \leftarrow \neg p, \neg q; \quad p \leftarrow p, q; \quad q \leftarrow p, \neg q; \quad q \leftarrow p, q\} \qquad \mathcal{P}_2 = \{p \leftarrow \neg p, \neg q; \quad p \leftarrow p, q; \quad q \leftarrow p\}$$

A closer look at the clauses 3 and 4 of $\mathcal{P}_1$ yields that q does not depend on q, hence we could replace those two clauses by the single clause $q \leftarrow p$, resulting in $\mathcal{P}_2$.

By generalizing from Examples 5.1 and 5.2, we introduce *α-reduced programs* in Definition 5.4. The following definition introduces q- and $\neg q$-subsumption, serving as abbreviations to keep the notions simple.

Definition 5.3. *A clause C_1 is said to q-subsume C_2, if we have:*

$$\begin{aligned} C_1 &= (a \leftarrow \ \ q, \underbrace{r_1, \ldots, r_a}_{\cap|}, \underbrace{\neg s_1, \ldots \neg s_b}_{\cap|}) \\ C_2 &= (a \leftarrow \neg q, \overbrace{t_1, \ldots, r_c}, \overbrace{\neg u_1, \ldots \neg u_d}). \end{aligned}$$

I.e., if C_1 (without q) subsumes C_2 (without $\neg q$) and we have additionally $q \in C_1$ and $\neg q \in C_2$. Analogously, C_1 is said to $\neg q$-subsume C_2, if:

$$\begin{aligned} C_1 &= (a \leftarrow \neg q, \underbrace{r_1, \ldots, r_a}_{\cap|}, \underbrace{\neg s_1, \ldots \neg s_b}_{\cap|}) \\ C_2 &= (a \leftarrow \ \ q, \overbrace{t_1, \ldots, r_c}, \overbrace{\neg u_1, \ldots \neg u_d}). \end{aligned}$$

Definition 5.4. *An α-reduced program $\mathcal{P}$ is a program satisfying:*

1. *$\mathcal{P}$ is reduced (Definition 2.4).*
2. *There are no two clauses $C_1 \neq C_2$ in $\mathcal{P}$ such that C_1 q-subsumes C_2.*
3. *There are no two clauses $C_1 \neq C_2$ in $\mathcal{P}$ such that C_1 $\neg q$-subsumes C_2.*

Both examples above (Example 5.1 and 5.2) show logic programs and their α-reduced versions. A method to construct an α-reduced logic program from an arbitrary program is given as Algorithm 5. Please note, that the first 3 reduction steps are the same as in Algorithm 1. The proof of the following proposition is straightforward but tedious, and is left to the reader.

Proposition 5.5. *Let $\mathcal{P}$ be a logic program. If $\mathcal{Q}$ is the result of Algorithm 5 on input $\mathcal{P}$, then $\mathcal{Q}$ is an α-reduced logic program and $T_{\mathcal{P}} = T_{\mathcal{Q}}$.*

Unfortunately, α-reduced programs are not necessarily minimal, as the next example shows.

Example 5.6. *The following two programs are equivalent. Even though both programs are α-reduced, P_3 is larger than P_2.*

$$\begin{aligned} \mathcal{P}_2 = \{ & p \leftarrow \neg p, \neg r; \\ & p \leftarrow p, r; \\ & p \leftarrow q, r \} \end{aligned} \qquad \begin{aligned} \mathcal{P}_3 = \{ & p \leftarrow \neg p, \neg r; \\ & p \leftarrow p, r; \\ & p \leftarrow q, r; \\ & p \leftarrow \neg p, q \} \end{aligned}$$

Note also that P_3 can be transformed to P_2 by removing a redundant clause. However, this cannot be done by α-reduction.

Algorithm 5: Constructing an α-reduced logic program

Input: An arbitrary program $\mathcal{P}$.
Output: An α-reduced logic program $\mathcal{Q}$ with $T_{\mathcal{P}} = T_{\mathcal{Q}}$.

1 Initialize $\mathcal{Q} = \mathcal{P}$.
2 **while** *one of the following reductions applies* **do**
3   **if** *there is an inconsistent clause C in $\mathcal{Q}$* **then**
4   Remove C.
5   **if** *there is a clause in $\mathcal{Q}$ whose body contains the literal L twice* **then**
6   Remove one occurrence of L from the body.
7   **if** *there are $C_1 \neq C_2$ in $\mathcal{Q}$ such that C_1 subsumes C_2* **then**
8   Remove C_2.
9   **if** *there are $C_1 \neq C_2$ in $\mathcal{Q}$ such that C_1 q-subsumes C_2* **then**
10   Remove $\neg q$ in the body of C_2.
11   **if** *there are $C_1 \neq C_2$ in $\mathcal{Q}$ such that C_1 $\neg q$-subsumes C_2* **then**
12   Remove q in the body of C_2.
13 Return $\mathcal{Q}$ as result

In a similar manner, we can refine α-reduction by introducing further refinement conditions. Refinement conditions can for example be obtained by recurring to insights from inverse resolution operators as used in Inductive Logic Programming [32]. Resulting algorithms, which we investigated, yield further refined programs at the cost of lower efficiency. The more refined algorithms return minimal programs with a higher probability. However, none of these algorithms can guarantee to obtain a minimal program, which is why we do not spell out this approach in more detail here. We rather accept the fact that we need to take a heuristic approach in order to obtain reasonable run-time behavior. We will discuss this next.

6 A Greedy Extraction Algorithm

We present another extraction algorithm for normal programs, which is closer in spirit to Algorithm 4 in that it incrementally builds a program. For this purpose, we introduce the notions of *valid* and *allowed clause bodies*, where the idea is that we do not want to allow clauses which clearly lead to an incorrect $T_{\mathcal{P}}$ operator, and we do not want to allow clauses, for which a shorter allowed clause exists.

Example 6.1 illustrates the intuition.

Example 6.1. *Let $T_{\mathcal{P}}$ be given as follows:*

$$T_{\mathcal{P}} = \{ \quad \emptyset \mapsto \{p\} \quad \{q\} \mapsto \{p\} \quad \{p,q\} \mapsto \emptyset \quad \{q,r\} \mapsto \{p\}$$
$$\{p\} \mapsto \emptyset \quad \{r\} \mapsto \emptyset \quad \{p,r\} \mapsto \{p\} \quad \{p,q,r\} \mapsto \{p\}\}$$

The 3 atoms p, q, r are being used, so there would be 27 different possible clause bodies, as shown in Table 1. The clause $p \leftarrow p$ is not valid, since we have $p \notin T_{\mathcal{P}}(\{p\})$, whereas $p \leftarrow p, q, r$ is valid but not allowed because $p \leftarrow p, r$ is valid and smaller.

We will give a formal definition of valid and allowed clauses, before continuing with the example. Please note that in the following definitions B is not necessarily a clause in $\mathcal{P}$.

Definition 6.2. *Let $T_{\mathcal{P}}$ be an immediate consequence operator, and h be a predicate. We call $B = p_1, \ldots, p_a, \neg q_1, \ldots, \neg q_b$* valid with respect to h and $T_{\mathcal{P}}$ *iff for every interpretation $I \subseteq B_{\mathcal{P}}$ with $I \models B$ we have $h \in T_{\mathcal{P}}(I)$.*

Definition 6.3. *Let $T_{\mathcal{P}}$ be an immediate consequence operator, and h be a predicate. We call $B = p_1, \ldots, p_a, \neg q_1, \ldots, \neg q_b$* allowed with respect to h and $T_{\mathcal{P}}$ *if the following two properties hold:*

1. *B is valid with respect to $T_{\mathcal{P}}$ and h.*
2. *There is no valid body $B' \subset B$ for h and $T_{\mathcal{P}}$.*

Example 6.4 (6.1 ctd.)**.** *Table 1 shows all possible clause bodies for $B_{\mathcal{P}} = \{p, q, r\}$. Furthermore it shows either "Allowed", if the body is allowed, or gives the reason why it is not allowed.*

Next, we will present an algorithm to compute the set of allowed clause bodies. The underlying ideas are illustrated with the help of a Hasse diagram (i.e., the partition of the Hasse diagram corresponding to consistent clause bodies) in Figure 4. First, all consistent clause bodies that contain every atom from $B_{\mathcal{P}}$ are constructed and those which are valid with respect to the $T_{\mathcal{P}}$-operator are included in a set $\mathcal{B}$. In Figure 4 those are depicted white in the bottom-row, while invalid clause bodies are marked gray. Then, for all those sets $B \in \mathcal{B}$ all direct subsets (subsets with one element less, i.e. one row up in the diagram) are constructed by removing one (possibly negated) atom b. Let $B' := B \setminus \{b\}$. If $B' \cup \{\neg b\}$ is also

Body	Evaluation
$\emptyset$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{p\}$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{q\}$	NV ($p \notin T_{\mathcal{P}}(\{p, q\})$)
$\{r\}$	NV ($p \notin T_{\mathcal{P}}(\{r\})$)
$\{\neg p\}$	NV ($p \notin T_{\mathcal{P}}(\{r\})$)
$\{\neg q\}$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{\neg r\}$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{p, q\}$	NV ($p \notin T_{\mathcal{P}}(\{p, q\})$)
$\{p, r\}$	Allowed
$\{q, r\}$	Allowed
$\{p, \neg q\}$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{p, \neg r\}$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{q, \neg p\}$	Allowed
$\{q, \neg r\}$	NV ($p \notin T_{\mathcal{P}}(\{p, q\})$)
$\{r, \neg p\}$	NV ($p \notin T_{\mathcal{P}}(\{r\})$)
$\{r, \neg q\}$	NV ($p \notin T_{\mathcal{P}}(\{r\})$)
$\{\neg p, \neg q\}$	NV ($p \notin T_{\mathcal{P}}\{r\})$)
$\{\neg p, \neg r\}$	Allowed
$\{\neg q, \neg r\}$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{p, q, r\}$	NA ($\{p, r\}$ is smaller)
$\{p, q, \neg r\}$	NV ($p \notin T_{\mathcal{P}}(\{p, q\})$)
$\{p, \neg q, r\}$	NA ($\{p, r\}$ is smaller)
$\{\neg p, q, r\}$	NA ($\{q, r\}$ is smaller)
$\{p, \neg q, \neg r\}$	NV ($p \notin T_{\mathcal{P}}(\{p\})$)
$\{\neg p, q, \neg r\}$	NA ($\{\neg p, q\}$ is smaller)
$\{\neg p, \neg q, r\}$	NV ($p \notin T_{\mathcal{P}}(\{r\})$)
$\{\neg p, \neg q, \neg r\}$	NA ($\{\neg p, \neg r\}$ is smaller)

Table 1: Evaluation of clause bodies for the $T_{\mathcal{P}}$-operator from Example 6.1. Bodies which are not valid are marked with "NV" and those that are not allowed with "NA". For both cases the reason is given in parentheses.

contained in $\mathcal{B}$ (i.e. white in the figure), we can conclude, that the truth value of b is not important. Therefore, we add the body B' to $\mathcal{B}$ and mark B and $B' \cup \{\neg b\}$ as *subsumed*. This process is repeated for all sets in $\mathcal{B}$. Finally, we return all valid elements from $\mathcal{B}$ that are not marked *subsumed*. The details can be found as Algorithm 6. In the sequel, we will show that the algorithm is sound and complete, i.e., that it returns all allowed clause bodies with respect to a given operator.

Lemma 6.5. *Let $T_{\mathcal{P}}$ and h be the input for Algorithm 6. Then every set B added to $\mathcal{B}$ is a valid clause body wrt. $T_{\mathcal{P}}$ and h.*

Proof sketch. Every clause body B added to $\mathcal{B}$ in line 5 of the algorithm is valid, because there is exactly one $I \subseteq B_{\mathcal{P}}$ such that $I \models B$ and we find $h \in T_{\mathcal{P}}(I)$ and hence B to be valid. Let B' be a body added to $\mathcal{B}$ in line 10. B' will be added iff $B := B' \cup \{b\}$ and $B'' = B' \cup \{\neg b\}$ are both valid. Let $\mathcal{I}$ be the set of interpretations mapping B to true; and let $\mathcal{I}'$ and $\mathcal{I}''$ be the sets of interpretations mapping B' and B'' to true, respectively. Obviously, we find $\mathcal{I} = \mathcal{I}' \cup \mathcal{I}''$, which completes the proof. $\square$

Figure 4: The Hasse diagram of consistent clause bodies over p, q, r. Valid clause bodies wrt. $T_{\mathcal{P}}$ from Example 6.1 are depicted in white, invalid ones gray.

Lemma 6.6. *Let $T_{\mathcal{P}}$ and h be the input for Algorithm 6. Then in line 12, $\mathcal{B}$ contains all valid clause bodies.*

Proof sketch. The following could be shown formally by induction on the size of the body D going from $|B_{\mathcal{P}}|$ to 1. Assume there is a valid clause body D which is not contained in $\mathcal{B}$. For $|D| = |B_{\mathcal{P}}|$ we derive a contradiction as D would have been included in line 5. For $|D| < |B_{\mathcal{P}}|$ with $q \notin D$ we can conclude that $D \cup \{q\}$ and $D \cup \{\neg q\}$ are also valid. Hence, they would be contained in $\mathcal{B}$ and we obtain a contradiction because D would have been added to $\mathcal{B}$ in line 10. Therefore we can conclude that $\mathcal{B}$ in line 12 contains all valid clause bodies. $\square$

Lemma 6.7. *Let $T_{\mathcal{P}}$ and h be the input, and $\mathcal{C}$ be the output of Algorithm 6. Then every set $C \in \mathcal{C}$ is an allowed clause body.*

Proof sketch. According to Definition 6.3, we find C to be allowed, if it is valid, and if (b) there is no valid $B' \subset B$. From Lemma 6.5, we know that every $C \in \mathcal{C}$ returned by Algorithm 6 is valid. Hence it remains to show that there is no valid $B' \subset B$. Assuming there would be a B' and $B \in \mathcal{C}$ with $B' \subseteq B$ leads immediately to a contradiction, because B would have been marked *subsumed* in line 11 and hence not be included in $\mathcal{C}$. $\square$

Proposition 6.8. *Algorithm 6 is correct and complete, i.e. it outputs the set of all allowed clause bodies with respect to a given $T_{\mathcal{P}}$-operator and an atom h.*

Proof sketch. Using Lemma 6.7, we know that every returned clause body is allowed, i.e., Algorithm 6 is correct. The completeness follows from Lemma 6.6 and the fact that every allowed clause body B is actually valid and not subsumed by some $B' \subseteq B$ which is ensured in line 11. $\square$

Algorithm 6: Computing allowed clause bodies

Input: An arbitrary mapping $f : I_{\mathcal{P}} \to I_{\mathcal{P}}$ over $B_P = \{q_1, \ldots, q_m\}$
Input: A target atom h
Output: The set of allowed clause bodies

```
1  Initialize 𝓑 = ∅.
2  foreach I := {a_1, ..., a_n} ⊆ B_𝒫 do
3      if h ∈ T_𝒫(I) then
4          Let {b_1, ..., b_l} := B_𝒫 \ I.
5          Add {a_1, ..., a_n, ¬b_1, ..., ¬b_l} to 𝓑.
6  foreach B added to 𝓑 (sorted descending wrt. their cardinality) do
7      foreach b ∈ B do
8          Let B' := B \ {b}.
9          if B' ∪ {¬b} ∈ 𝓑 then
10             Add B' to 𝓑.
11             Mark B and B' ∪ {¬b} to be subsumed.
12 Return 𝓑 \ {B ∈ 𝓑 | B is marked subsumed} as result.
```

We use the notion of allowed clause bodies to present a greedy algorithm that constructs a logic program for a given target function. The algorithm will incrementally add clauses to an initially empty program. The clause to add is chosen from the set of allowed clauses with respect to some *score*-function, which is a heuristics for the importance of a clause. This function computes the number of interpretations for which the program does not yet behave correctly, but for which it would after adding the clause.

Definition 6.9. *Let $B_{\mathcal{P}}$ be a set of predicates. The* score *of a clause $C : h \leftarrow \mathrm{B}$ with respect to a program $\mathcal{P}$ is defined as*

$$\mathrm{score}(C, \mathcal{P}) := \big|\{I \mid I \subseteq B_{\mathcal{P}} \textit{ and } h \notin T_{\mathcal{P}}(I) \textit{ and } I \models \mathrm{B}\}\big|.$$

Please note, that the score-function can easily be implemented based on insights from the Hasse-diagrams as depicted in Figure 4.

To keep things simple, we will consider one predicate at a time only, since after treating every predicate symbol, we can put the resulting sub-programs together. Let $q \in B_{\mathcal{P}}$ be an atom, then we call $T_{\mathcal{P}}^q$ the *restricted* consequence operator for q

Algorithm 7: Greedy Extraction Algorithm

Input: An arbitrary mapping $f : I_{\mathcal{P}} \to I_{\mathcal{P}}$ over $B_P = \{q_1, \ldots, q_m\}$
Output: A logic program $\mathcal{Q}$ with $T_{\mathcal{Q}} = f$.

1 Initialize $\mathcal{Q} = \emptyset$.
2 **foreach** *predicate* $q_i \in B_{\mathcal{P}}$ **do**
3   Construct the set S_i of allowed clause bodies for q_i.
4   Let $\mathcal{C} := \{q_i \leftarrow B \mid B \in S_i\}$.
5   Initialize: $\mathcal{Q}_i = \emptyset$.
6   **repeat**
7   Let $s := max_{C \in \mathcal{C}}\,(\text{score}(C, Q_i))$.
8   Let $\mathcal{C}' := \{C \mid C \in \mathcal{C}$ and $\text{score}(C, Q_i) = s\}$.
9   Add C to $\mathcal{Q}_i$, with C being one of the smallest clauses in $\mathcal{C}'$.
10   **until** $T_{\mathcal{Q}_i} = T_{\mathcal{P}}^{q_i}$ *(*$s = 0$*)*
11   Add $\mathcal{Q}_i$ to $\mathcal{Q}$.
12 Return $\mathcal{Q}$ as result

and set $T_{\mathcal{P}}^{q}(I) = \{q\}$ if $q \in T_{\mathcal{P}}(I)$, and $T_{\mathcal{P}}^{q}(I) = \emptyset$ otherwise. Algorithm 7 gives the details of the resulting procedure and is illustrated in Example 6.10.

Example 6.10. *Let $T_{\mathcal{P}}$ be given as follows:*

$$T_{\mathcal{P}} = \left\{ \begin{array}{cccc} \emptyset \mapsto \{p\} & \{r\} \mapsto \emptyset & \{q,r\} \mapsto \emptyset & \{p,q,s\} \mapsto \emptyset \\ \{p\} \mapsto \emptyset & \{p,q\} \mapsto \{p\} & \{q,s\} \mapsto \{p\} & \{p,r,s\} \mapsto \{p\} \\ \{q\} \mapsto \{p\} & \{p,r\} \mapsto \{p\} & \{r,s\} \mapsto \emptyset & \{q,r,s\} \mapsto \{p\} \\ \{r\} \mapsto \emptyset & \{p,s\} \mapsto \{p\} & \{p,q,r\} \mapsto \{p\} & \{p,q,r,s\} \mapsto \{p\} \end{array} \right\}$$

Obviously, we can concentrate on the predicate p, since there are no other predicates occurring as a consequence. The resulting set of allowed clause bodies is

$$S = \big\{(p,r); (\neg p, \neg r, \neg s); (q, \neg p, \neg r); (q, \neg r, \neg s); \\ (p, q, \neg s); (p, s, \neg q); (q, s, \neg p); (q, r, s)\big\}$$

Tables 2 and 3 show two example runs of Algorithm 7. In each step the score for the allowed clauses which are not yet in the program, is indicated. (The score of the clause which is added to the constructed program $\mathcal{Q}$ is given in boldface.) E.g., the score for $p, q, \neg s$ in the first step of the first run is 2, because

clause body	1.	2.	3.	4.	5.	6.
p, r	$\boxed{\mathbf{4}}$					
$\neg p, \neg r, \neg s$	2	2	$\boxed{\mathbf{1}}$			
$q, \neg p, \neg r$	2	$\boxed{\mathbf{2}}$				
$q, \neg r, \neg s$	2	2	1	$\boxed{\mathbf{1}}$		
$p, q, \neg s$	2	1	1	1	0	0
$p, s, \neg q$	2	1	1	1	$\boxed{\mathbf{1}}$	
$q, s, \neg p$	2	2	1	1	1	$\boxed{\mathbf{1}}$
q, r, s	2	1	1	1	1	1

$$\begin{aligned}\mathcal{P}_1 = \{p &\leftarrow p, r;\\ p &\leftarrow q, \neg p, \neg r;\\ p &\leftarrow \neg p, \neg r, \neg s;\\ p &\leftarrow q, \neg r, \neg s;\\ p &\leftarrow p, s, \neg q;\\ p &\leftarrow q, s, \neg p\}\end{aligned}$$

Table 2: Example run 1 and the resulting program.

clause body	1.	2.	3.	4.	5.
p, r	$\boxed{\mathbf{4}}$				
$\neg p, \neg r, \neg s$	2	$\boxed{\mathbf{2}}$			
$q, \neg p, \neg r$	2	2	1	0	0
$q, \neg r, \neg s$	2	2	1	$\boxed{\mathbf{1}}$	
$p, q, \neg s$	2	1	1	1	0
$p, s, \neg q$	2	1	1	1	$\boxed{\mathbf{1}}$
$q, s, \neg p$	2	2	$\boxed{\mathbf{2}}$		
q, r, s	2	1	1	0	0

$$\begin{aligned}\mathcal{P}_2 = \{p &\leftarrow p, r;\\ p &\leftarrow \neg p, \neg r, \neg s;\\ p &\leftarrow q, s, \neg p;\\ p &\leftarrow q, \neg r, \neg s;\\ p &\leftarrow p, s, \neg q\}\end{aligned}$$

Table 3: Example run 2 and the resulting program.

$p \in T_{\mathcal{P}}(\{p, q\})$ *and* $p \in T_{\mathcal{P}}(\{p, q, r\})$*. It goes down to 1 in the second step, because we added* $p \leftarrow p, r$ *to* $\mathcal{Q}$ *and only* $p \in T_{\mathcal{Q}}(\{p, q, r\})$ *is left. This means that we would only gain one additional interpretation by adding* $p \leftarrow p, q, \neg s$ *as second clause.*

For Example 6.10 there are two different possible runs of the algorithm, which return programs of different size for the same operator. The first run produces a program with six clauses and 17 literals. The second run produces a program with five clauses and 14 literals. This shows that the algorithm does not always return a minimal program, which was expected since the algorithm is greedy, i.e. it chooses the clause with respect to some heuristics and without forecasting the effects of this decision. We also see that the algorithm is not deterministic, because there may be several clauses with the highest score and the lowest number of literals (e.g. in step 3 of run 1).

While it is clear that the score function is a meaningful heuristics, it is difficult

		valid bodies		allowed bodies	
n	3^n	max	avg / red	max	avg / red
2	9	9	3.0 / 36%	2	1,3 / 15%
3	27	27	7.3 / 27%	6	2,5 / 9%
4	81	51	17.1 / 21%	13	5,2 / 6%
5	243	119	40.6 / 17%	20	11,1 / 5%
6	729	187	96.5 / 13%	41	24,0 / 3%
7	2,187	409	219.9 / 10%	73	52,8 / 2%
8	6,561	800	501.7 / 8%	158	117,5 / 2%
9	19,683	1,548	1,136.9 / 6%	329	261,8 / 1%
10	59,049	3,259	2,570.4 / 4%	688	583,8 / 1%

Table 4: Evaluation of clause pruning.

to judge the performance of the pruning method, i.e. the computation of allowed clauses. To evaluate the pruning we ran several tests, the results of which are shown in Table 4. The left-most column shows the number of predicates, which were used for the test. For this number of predicates we randomly generated T_P^q operators for a fixed predicate q and computed the set of allowed clauses for this operator. We are, of course, interested in the number of allowed clauses we get, because this gives us an idea how good the pruning method works. To get a good approximation we generated 1000 operators for each fixed number of predicates. The second column in Table 4 shows the number of all clause bodies, which exist using n predicates (this is 3^n). The third column shows the maximum number of valid clause bodies over all 1000 samples, followed by the average number and the corresponding reduction ratio wrt. the total number of clause bodies. Finally, the same numbers are given for allowed clause bodies.

A look at the table reveals that the number of existing clause bodies grows by a factor of 3 if a predicate is added, whereas the maximum and average in the samples grow approximately by a factor of 2. As a consequence the percentage of allowed clauses will decrease with an increasing number of predicates, i.e. only a smaller fraction of clauses survive the pruning process. In general a good reduction is achieved, which in the next section will allow us to define an *Intelligent Program Search Algorithm*. It is important to notice that the algorithm will probably perform even better in real world tasks, because there the T_P operator usually follows certain patterns, which tends to lead to shorter rules. A consequence is that the number of allowed clauses is likely to be lower than the average in column 4.

Also note that the number of allowed clauses is an upper bound for the size of the output program, which is returned by the greedy algorithm.

7 Extracting Minimal Normal Programs

The discussed approaches for extracting normal programs cannot guarantee minimality of the extracted program. This raises the following question: *Is there always a unique minimal (i.e. least) program for any given target function?* The answer is negative, as the following proposition shows.

Proposition 7.1. *There can be more than one minimal propositional logic program with a given immediate consequence operator.*

Proof. For the programs P_1 and P_2 below (corresponding to the operator in example 6.1), there is no smaller program, i.e. a program with less literals, and the same immediate consequence operator.

$$\begin{aligned} \mathcal{P}_1 = \{p &\leftarrow \neg p, \neg r; \\ p &\leftarrow p, r; \\ p &\leftarrow \neg p, q\} \end{aligned} \qquad\qquad \begin{aligned} \mathcal{P}_2 = \{p &\leftarrow \neg p, \neg r; \\ p &\leftarrow p, r; \\ p &\leftarrow q, r\} \end{aligned}$$

To prove this we have to show that there is no smaller program having the operator $T_P := T_{P_1} = T_{P_2}$. In Example 6.1 we have computed the set S of allowed clause bodies for the predicate p (Obviously, a minimal program with this operator consists of clauses with head p only): $S = \{p, r; q, r; q, \neg p; \neg p, \neg r\}$. In particular there is no clause body with one or three literals. This means for a program to be smaller than P_1 or P_2, which have the size of nine literals, it must have one or two clauses where each clause has exactly three literals (exactly two literals in the body). Furthermore, a program must contain the clause $p \leftarrow \neg p, \neg r$, because it is the only one for which we get $T_P(\emptyset) = \{p\}$. We also need the clause $p \leftarrow p, r$, because it is the only one which gives us $T_P(\{p, r\}) = \{p\}$. But for the program $Q = \{p \leftarrow \neg p, \neg r; p \leftarrow p, r\}$ we have $p \notin T_Q(\{q, r\})$. This means that there is no program with two or less clauses with operator T_P. This completes the proof. $\square$

Proposition 7.1 shows that an analogy to Corollary 4.4 does not hold for normal programs. This means that at best we can hope to extract *minimal* normal programs from neural networks, but in general not a *least* program. The complexity of this task is yet unknown, as is an optimal extraction algorithm. However,

Algorithm 8: Intelligent Program Search

Input: An arbitrary mapping $f : I_{\mathcal{P}} \to I_{\mathcal{P}}$ over $B_P = \{q_1, \dots, q_m\}$
Output: A logic program $\mathcal{Q}$ with $T_{\mathcal{Q}} = f$.

1 Initialize $\mathcal{Q} = \emptyset$.
2 **foreach** *predicate* $q_i \in B_{\mathcal{P}}$ **do**
3 Construct the set S_i of allowed clause bodies for q_i.
4 Initialize: $n_i = 0$
5 **repeat**
6 **if** *there is no allowed program* $\mathcal{Q}_i$ *with* $T_{\mathcal{P}}^{q_i} = T_{\mathcal{Q}_i}$ *and size* n_i **then**
7 Increment n_i
8 **until** *until an allowed program* $\mathcal{Q}_i$ *was found*
9 Add $\mathcal{Q}_i$ to $\mathcal{Q}$.
10 Return $\mathcal{Q}$ as result

we can modify Algorithm 7 in order to obtain minimal programs. We do this by performing a full program search instead of using a heuristics, i.e. the score function, to add clauses to subprograms. The resulting Algorithm 8 is more intelligent than a naive full program search algorithm in that it constructs subprograms for each predicate separately and makes use of the pruning method introduced for the greedy algorithm.

8 Experimental Evaluation

Even though we focused on a theoretical investigation, we would like to present some experimental results. All algorithms described above were implemented in Java 1.5 and applied to some (benchmark) problems. First we will present a very small example in detail. Afterwards, solutions for three problems from the *UCI Machine Learning Repository* [12] are presented. These are the *Monks Problem 1 and 2* and the *Shuttle Landing Control* problem.

We trained a neural network with data about *Nessie* from Loch Ness. The learned input-output mapping is given in Table 5. We used the following abbreviations:

input					output				
a	d	f	i	t	a	d	f	i	t
+	+	+	+	+	-	+	-	+	+
+	+	+	+	-	-	+	-	+	+
+	+	+	-	+	-	+	-	+	+
+	+	+	-	-	-	+	-	+	+
+	+	-	+	+	+	+	-	-	+
+	+	-	+	-	+	+	-	-	+
+	+	-	-	+	+	+	-	-	+
+	+	-	-	-	+	+	-	-	+
+	-	+	+	+	-	+	-	+	-
+	-	+	+	-	-	+	-	+	-
+	-	+	-	+	-	+	-	+	-
+	-	+	-	-	-	+	-	+	-
+	-	-	+	+	+	+	-	-	-
+	-	-	+	-	+	+	-	-	-
+	-	-	-	+	+	+	-	-	-
+	-	-	-	-	+	+	-	-	-

input					output				
a	d	f	i	t	a	d	f	i	t
-	+	+	+	+	-	+	-	+	+
-	+	+	+	-	-	+	-	+	+
-	+	+	-	+	-	-	-	+	+
-	+	+	-	-	-	-	-	+	+
-	+	-	+	+	+	+	-	-	+
-	+	-	+	-	+	+	-	-	+
-	+	-	-	+	+	-	-	-	+
-	+	-	-	-	+	-	-	-	+
-	-	+	+	+	-	+	-	+	-
-	-	+	+	-	-	+	-	+	-
-	-	+	-	+	-	-	-	+	-
-	-	+	-	-	-	-	-	+	-
-	-	-	+	+	+	+	-	-	-
-	-	-	+	-	+	+	-	-	-
-	-	-	-	+	+	-	-	-	-
-	-	-	-	-	+	-	-	-	-

Table 5: Input-output mapping for the fairy-tale example.

f *nessie is a fairy tale creature*,
i *nessie is immortal*,
a *nessie is an animal*,
d *nessie is a dragon* and
t *nessie is a tourist attraction*.

One, obvious implication in that data, which will hopefully be reflected in the resulting program, is $t \leftarrow d$, because we find a "+" for the output-t, whenever there is a "+" for the input-d.

Applying Algorithm 3, we obtain a program consisting of 72 clauses, namely one for each "+" on the output side, where the body represents the corresponding input. The partial result for the head d, is shown in Figure 5. After applying the α-reduction (Algorithm 5), we obtain the program shown in Figure 6. This program was actually the program we used to generate samples to train the neural network. Algorithm 6 returned 5 allowed clause bodies. The Greedy algorithm as well as the intelligent program search returned the same final program. In Table 7 the number of clause bodies is given and Table 8 shows the running times of the different algorithms.

Next, we studied the Monks problems [36]. These are classification problems, where monks must be classified according to six attributes. Those attributes are

$$
\begin{array}{lll}
d \leftarrow a, d, f, i, t. & d \leftarrow a, d, f, i, \neg t. & d \leftarrow a, d, f, \neg i, t. \\
d \leftarrow a, d, f, \neg i, \neg t. & d \leftarrow a, d, \neg f, i, t. & d \leftarrow a, d, \neg f, i, \neg t. \\
d \leftarrow a, d, \neg f, \neg i, t. & d \leftarrow a, d, \neg f, \neg i, \neg t. & d \leftarrow a, \neg d, f, i, t. \\
d \leftarrow a, \neg d, f, i, \neg t. & d \leftarrow a, \neg d, f, \neg i, t. & d \leftarrow a, \neg d, f, \neg i, \neg t. \\
d \leftarrow a, \neg d, \neg f, i, t. & d \leftarrow a, \neg d, \neg f, i, \neg t. & d \leftarrow a, \neg d, \neg f, \neg i, t. \\
d \leftarrow a, \neg d, \neg f, \neg i, \neg t. & d \leftarrow \neg a, d, f, i, t. & d \leftarrow \neg a, d, f, i, \neg t. \\
d \leftarrow \neg a, d, \neg f, i, t. & d \leftarrow \neg a, d, \neg f, i, \neg t. & d \leftarrow \neg a, \neg d, f, i, t. \\
d \leftarrow \neg a, \neg d, f, i, \neg t. & d \leftarrow \neg a, \neg d, \neg f, i, t. & d \leftarrow \neg a, \neg d, \neg f, i, \neg t.
\end{array}
$$

Figure 5: Partial result (definition of d) of Algorithm 3, if applied to the mapping shown in Table 5.

$i \leftarrow f.$	*// if nessie is a fairy tale creature then she is immortal*
$a \leftarrow \neg f.$	*// if nessie is not a fairy tale creature then she is an animal*
$d \leftarrow a.$	*// if nessie is an animal then she is a dragon*
$d \leftarrow i.$	*// if nessie is immortal then she is a dragon*
$t \leftarrow d.$	*// if nessie is a dragon then she is a tourist attraction*

Figure 6: The α-reduced (Algorithm 5) version of the result of Algorithm 3, if applied to the mapping shown in Table 5.

represented using one or two propositional variables each, as shown in Table 6.

For each of the Monks Problems, a set of attribute-class pairs is given, similar to Table 5, but with only one output called $class$. In problem 1, we find $class = 1$ whenever the value of attribute a and b coincide, or if $e = 1$. The set of 432 samples can easily be represented as a propositional logic program using the encoding shown in Table 6 and adding a corresponding clause for each item with $class = 1$. Applying Algorithm 5, we obtain the program shown in Figure 7. As before, the outputs of the Greedy algorithm coincides with that result.

For the second Monks Problem, we find $class = 1$ whenever exactly two attributes have the value 1. The resulting program for the Monks Problem 2 is partly shown in Figure 8. Unfortunately, and mainly due to our encoding, the result

Att.	Values	Propositional encoding of the different values 1	2	3	4
a	$\{1,2,3\}$	$\{a_1, a_2\}$	$\{a_1, \neg a_2\}$	$\{\neg a_1\}$	-
b	$\{1,2,3\}$	$\{b_1, b_2\}$	$\{b_1, \neg b_2\}$	$\{\neg b_1\}$	-
c	$\{1,2\}$	$\{c_1\}$	$\{\neg c_1\}$	-	-
d	$\{1,2,3\}$	$\{d_1, d_2\}$	$\{d_1, \neg d_2\}$	$\{\neg d_1\}$	-
e	$\{1,2,3,4\}$	$\{e_1, e_2\}$	$\{e_1, \neg e_2\}$	$\{\neg e_1, e_2\}$	$\{\neg e_1, \neg e_2\}$
f	$\{1,2\}$	$\{f_1\}$	$\{\neg f_1\}$	-	-

Table 6: A propositional encoding of attribute-values for the Monks Problem.

$monk_1 \leftarrow a_1, a_2, b_1, b_2.$ // $monk_1$ *if* $a = 1$ *and* $b = 1$

$monk_1 \leftarrow a_1, b_1, \neg a_2, \neg b_2.$ // $monk_1$ *if* $a = 2$ *and* $b = 2$

$monk_1 \leftarrow \neg a_1, \neg b_1.$ // $monk_1$ *if* $a = 3$ *and* $b = 3$

$monk_1 \leftarrow e_1, e_2.$ // $monk_1$ *if* $e = 1$

Figure 7: The resulting program for the Monks Problem 1.

consists of 104 clauses, because we can not use an explicit negation. Therefore, the program search does not terminate, as 2^{104} clauses would have to be tested.

As a last experiment, we used the Shuttle Landing Control data set from the UCI repository. Using 6 attributes, it must be decided, whether to use manual or automatic control for the landing procedure of some shuttle. As for the Monks Problems, we encoded the attributes using propositional variables and the full dataset as initial program. The alpha reduced version of this program contains 33 literals, which can be further reduced to 21 literals using the allowed clauses.

Table 7 shows the total number of clauses, the number of valid and the number of allowed clauses for each of the four experiments, together with the corresponding ratios. Furthermore, we found that computing the allowed clause bodies actually solved our problems. This is not necessarily the case as indicated by Example 6.10, i.e., there might be redundant allowed clauses for a given operator. Table 8 shows the different running times of the algorithms for all four experiments. Most of the time was needed to compute the allowed clause bodies, which is not surprising as this basically solved the problem.

This experiments are not meant to be exhaustive, but to show that the algo-

$$monk_2 \leftarrow a_1, a_2, b_1, b_2, \neg c_1, \neg d_1, \neg e_1, \neg f_1.$$
$$monk_2 \leftarrow a_1, a_2, b_1, b_2, \neg c_1, \neg d_1, \neg e_2, \neg f_1.$$
$$monk_2 \leftarrow a_1, a_2, b_1, b_2, \neg c_1, \neg d_2, \neg e_1, \neg f_1.$$
$$monk_2 \leftarrow \underbrace{a_1, a_2}_{a=1}, \underbrace{b_1, b_2}_{b=1}, \neg c_1, \neg d_2, \neg e_2, \neg f_1.$$

Figure 8: Part of the resulting program for the Monks Problem 2. The four clauses represent the fact that $a = b = 1$ and none of the other attributes has value 1.

Clause-type	Nessie (%)	Monk1 (%)	Monk2 (%)	Shuttle (%)
Possible	1,215 (100)	59,049 (100)	59,049 (100)	6,561 (100)
Valid	243 (20)	13,689 (23)	1,775 (3)	1,363 (20)
Allowed	5 (0.4)	4 (0.007)	104 (0.2)	5 (0.08)

Table 7: Number of clauses occurring in the experiments.

rithms presented here actually work. We showed, how to obtain a reduced logic program from a given input-output mapping. This was done by first converting the mapping into a "complete" logic program which was reduced afterwards. In one aspect the experiments exceeded our expectations. Namely in the effectiveness of the definition of allowed clauses. Even though the pruning results presented in Table 4 indicated a good average performance, the effect on the "real-world" problems was even bigger. For the Nessie-problem, only 0.4% of all clauses are actually allowed (Monk1: 0.007%, Monk2: 0.2%, Shuttle: 0.08%). In our examples, the problems were already solved by computing the allowed clauses. Using Algorithm 6 we have an efficient way to compute these allowed clauses without evaluating the complete $T_{\mathcal{P}}$ operator for each clause body to analyse.

Overall, all algorithms solve the problems in reasonable time and in most cases were able to extract a minimal program (the exception is α-reduction on the Shuttle Landing Control example). Again, we want to stress that we consider our results to be of fundamental nature, which means that applying the ideas to very large problems is likely to require approximations, e.g. by dropping the correctness guarantee.

	Nessie	Monk1	Monk2	Shuttle
α-reduction (Alg. 5)	25	2,040	900	360
Allowed Clauses (Alg. 6)	100	5,250	4,450	440
Greedy (Alg. 7)	4	56	700	10
Intelligent Program Search (Alg. 8)	1	7	?	4

Table 8: Running times (average over 5 runs) in ms of the different algorithms. We used a preliminary implementation in Java 1.5 on a 1.66 GHz PC with 512MB RAM. The program search for Monk2 did not terminate, as 2^{104} different clauses need to be tested. The times for Greedy and Intelligent Program Search are without the time needed for generating the allowed clause bodies.

9 Related Work

This work should be understood as part of broader investigations concerning the realization of the neural-symbolic learning cycle depicted in Figure 1. Work on different aspects of this cycle is historically done from a logic programming perspective.

This line of investigation was spawned in [25] by showing that every logic program can be implemented using a 3-layer network of binary threshold units, and that 2-layer networks do not suffice. It was also shown that under some syntactic restrictions on the programs, their semantics could be recovered by recurrently connecting the output- and the input layer of the network and propagating activation exhaustively through the resulting recurrent network. The key idea to [25] was to represent logic programs by means of their associated semantic operators instead of encoding the program directly, i.e. the functional input-output behavior of a semantic operator T_P associated with a given program P is encoded by means of a feedforward neural network N_P which, when presented an encoding of some I to its input nodes, produces $T_P(I)$ at its output nodes. This representation paradigm also underlies our work in this paper. Output nodes can also be connected recurrently back to the input nodes, resulting in a connectionist computation of iterates of I under T_P, as used e.g. in the computation of the semantics or meaning of P [31]. The relevance of this rather close relation to [25] lies in the idea of combining learning and reasoning within connectionist systems: While our work here focusses on understanding the knowledge implicit in a trained network, [25] is about reasoning *within the same representation paradigm*. As such, our investigations serve the long-term vision of realising autonomous systems which

can learn and reason while acting in changing environments.

This idea for the representation of logic programs spawned several investigations in different directions. As [25] employed binary threshold units as activation functions of the network nodes, the results were lifted to sigmoidal and hence differentiable activation functions in [21, 17]. This way, the connectionist representation of logic programs resulted in a network architecture which could be trained using standard backpropagation algorithms. The resulting connectionist inductive learning and reasoning system CILP was completed by providing corresponding knowledge extraction algorithms [14]. Further extensions to this include modal [20] and intuitionistic logics [16]. Metalevel priories between rules were introduced in [18]. An in-depth treatment of the whole approach can be found in [15]. The *knowledge based artificial neural networks (KBANN)* [38] are closely related to this approach, by using similar techniques to implement propositional logic formulae within neural networks, but with a focus on learning.

Another work following up on [25] concerns the connectionist treatment of first-order logic programming. [34] and [35] approach this by approximating given first-order programs P by finite subprograms of the grounding of P. These subprograms can be viewed as propositional ones and encoded using the original algorithm from [25]. [34] and [35] show that arbitrarily accurate encodings are possible for certain programs including definite ones (i.e. programs not containing negation as failure). They also lift their results to logic programming under certain multi-valued logics [28].

A more direct approach to the representation of first-order logic programs based on [25] was pursued in [26, 22, 23, 5, 9, 4]. The basic idea again is to represent semantic operators $T_P : I_P \to I_P$ instead of the program P directly. In [25] this was achieved by assigning propositional variables to nodes, whose activations indicate whether the nodes are true or false within the currently represented interpretation. In the propositional setting this is possible because for any given program only a finite number of truth values of propositional variables plays a role – and hence the finite network can encode finitely many propositional variables in the way indicated. For first-order programs, infinite interpretations have to be taken into account, thus an encoding of ground atoms by one neuron each is impossible as it would result in an infinite network, which is not computationally feasible to work with.

The solution put forward in [26] is to employ the capability of standard feedforward networks to propagate real numbers. The problem is thus reduced to encoding I_P as a set of real numbers in a computationally feasible way, and to provide means to actually construct the networks starting from their input-output be-

havior. Since sigmoidal units can be used, the resulting networks are trainable by backpropagation. [26] spelled out these ideas in a limited setting for a small class of programs, and was lifted in [22] to a more general setting, including the treatment of multi-valued logics. [23] related the results to logic programming under non-monotonic semantics. In these reports, it was shown that approximation of logic programs by means of standard feedforward networks is possible up to any desired degree of accuracy, and for fairly general classes of programs. However, no algorithms for practical generation of approximating networks from given programs could be presented. This was finally done in [9, 4], where also experiments are reported on which show the feasibility of the approach.

There exist two alternative approaches to the representation of first-order logic programs via their semantic operators, which have not been studied in more detail yet. The first approach, reported in [6], uses insights from fractal geometry as in [10] to construct iterated function systems whose attractors correspond to graphs of the semantic operators. The second approach builds on *Fibring logics* [13], and the corresponding Fibring Neural Networks [19]. The resulting system, presented in [5] and extended in [29], employs the fibring idea to control the firing of nodes such that it corresponds to term matching within a logic programming system. It is shown that certain limited kinds of first-order logic programs can be encoded this way, such that their models can be computed using the network.

Apart from the more general perspective on our work by means of the neural-symbolic learning cycle, this paper addresses the particular point of extracting propositional logic programs from artificial neural networks, and as such is closely related to the respective part of CILP, which was presented in [14]. The latter also addresses heuristics, as we do in the part on normal logic programs, for extracting propositional programs from networks. The perspective and approach taken, however, is entirely different. While CILP analyzes the structure of the network before extracting, we take a globalist view and refine the program. Regretfully, the CILP system is no longer available for an exhaustive experimental comparison.

There is also a considerable body of work on extracting knowledge from recurrent neural networks, usually in the form of finite state machines. This work is not easily comparable to our own, and it shall suffice to point the interested reader to the review article [27].

10 Conclusions

We presented algorithms for extracting definite and normal propositional logic programs from neural networks. For the case of definite programs, we have shown that our algorithm is optimal in the sense that it yields the least program with the desired operator; and it was formally shown that such a least program always exists. For normal logic programs we presented algorithms for obtaining minimal programs, and more efficient algorithms which do produce small but not necessarily minimal programs.

The main contribution of this paper is threefold.

- We investigated the existence of least respectively minimal extracted programs, including a formal proof that least definite programs always exist.
- We have given concrete extraction algorithms, and have shown by evaluations that they are feasible.
- We have addressed and answered fundamental (and obvious) open questions on the extraction of reduced logic programs from artificial neural networks.

We consider the results as a base for investigating the extraction of first-order logic programs, and thus for the development of the neural-symbolic learning cycle as laid out in Figure 1, which has high potential for impact in a variety of application areas.

References

[1] R. Alexandre, J. Diederich, and A. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge Based Systems*, pages 373–389, 1995.

[2] R. Andrews and S. Geva. Rule extraction from local cluster neural nets. *Neurocomputing*, 47(1–4):1–20, 2002.

[3] S. Bader, P. Hitzler, and S. Hölldobler. The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. In L. Li and K.K. Yen, editors, *Proceedings of the Third International Conference on Information*, pages 22–33, Tokyo, Japan, November/December 2004. International Information Institute.

[4] S. Bader, P. Hitzler, and S. Hölldobler. Connectionist model generation: A first-order approach. *Neurocomputing*, 2008. To appear.

[5] Sebastian Bader, Artur Garcez, and Pascal Hitzler. Computing first-order logic programs by fibring artificial neural networks. In I. Russell and Z. Markov, editors, *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Symposium Conference, Clearwater Beach, Florida, USA*, pages 314–319. AAAI Press, 2005.

[6] Sebastian Bader and Pascal Hitzler. Logic programs, iterated function systems, and recurrent radial basis function networks. *Journal of Applied Logic*, 2(3):273–300, 2004.

[7] Sebastian Bader and Pascal Hitzler. Dimensions of neural-symbolic integration – a structured survey. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 1, pages 167–194. International Federation for Computational Logic, College Publications, 2005.

[8] Sebastian Bader, Pascal Hitzler, Steffen Hölldobler, and Andreas Witzel. The core method: Connectionist model generation for first-order logic programs. In Barbara Hammer and Pascal Hitzler, editors, *Perspectives of Neural-Symbolic Integration*, volume 77 of *Studies in Computational Intelligenc*, pages 205–232. Springer, Berlin, 2007.

[9] Sebastian Bader, Pascal Hitzler, Steffen Hölldobler, and Andreas Witzel. A fully connectionist model generator for covered first-order logic programs. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 666–671, 2007.

[10] Michael Barnsley. *Fractals Everywhere*. Academic Press, San Diego, CA, USA, 1993.

[11] Ch. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[12] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998.

[13] Dov M. Gabbay. *Fibring Logics*. Oxford Univesity Press, 1999.

[14] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 126(1–2):155–207, 2001.

[15] A. S. d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems — Foundations and Applications*. Perspectives in Neural Computing. Springer, Berlin, 2002.

[16] A. S. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay. Neural-symbolic intuitionistic reasoning. In M. Koppen A. Abraham and K. Franke, editors, *Frontiers in Artificial Intelligence and Applications*, Melbourne, Australia, December 2003. IOS Press. Proceedings of the Third International Conference on Hybrid Intelligent Systems (HIS'03).

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

[18] A.S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. Metalevel priorities and neural networks. In *Proceedings of the Workshop on the Foundations of Connectionist-Symbolic Integration, ECAI'2000, Berlin*, August 2000.

[19] A.S. d'Avila Garcez and Dov M. Gabbay. Fibring neural networks. In *In Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 04). San Jose, California, USA, July 2004*. AAAI Press, 2004.

[20] A.S. d'Avila Garcez, L. C. Lamb, and D.M. Gabbay. A connectionist inductive learning system for modal logic programming. In *Proceedings of the IEEE International Conference on Neural Information Processing ICONIP'02, Singapore*, 2002.

[21] A.S. d'Avila Garcez, Gerson Zaverucha, and Luis A. V. de Carvalho. Logical inference and inductive learning in artificial neural networks. In Christoph Hermann, Frank Reine, and Antje Strohmaier, editors, *Knowledge Representation in Neural networks*, pages 33–46. Logos Verlag, Berlin, 1997.

[22] P. Hitzler, S. Hölldobler, and A. K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.

[23] Pascal Hitzler. Corollaries on the fixpoint completion: studying the stable semantics by means of the Clark completion. In D. Seipel, M. Hanus,

U. Geske, and O. Bartenstein, editors, *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management and the 18th Workshop on Logic Programming, Potsdam, Germany, March 4-6, 2004*, volume 327 of *Technichal Report*, pages 13–27. Bayerische Julius-Maximilians-Universität Würzburg, Institut für Informatik, 2004.

[24] Pascal Hitzler, Sebastian Bader, and Artur Garcez. Ontology leaning as a use case for neural-symbolic integration. In *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05*, 2005.

[25] S. Hölldobler and Y. Kalinke. Towards a new massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.

[26] S. Hölldobler, Y. Kalinke, and H.-P. Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999.

[27] H. Jacobsson. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Computation*, 17(6):1223–1263, 2005.

[28] E. Komendantskaya, A.K. Seda, and V. Komendantsky. On approximation of the semantic operators determined by bilattice-based logic programs. In *Proceedings of the Seventh International Workshop on First-Order Theorem Proving (FTP'05), Koblenz, Germany, September 2005*, pages 112–130, 2005.

[29] V. Komendantsky and A.K. Seda. Computation of normal logic programs by fibring neural networks. In *Proceedings of the Seventh International Workshop on First-Order Theorem Proving (FTP'05), Koblenz, Germany, September 2005*, pages 97–111, 2005.

[30] F. J. Kurfess. Neural networks and structured knowledge: Rule extraction and applications. *Applied Intelligence*, 12(1–2):7–13, 2000.

[31] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1988.

[32] S.H. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[33] J.-F. Remm and F. Alexandre. Knowledge extraction using artificial neural networks: application to radar target identification. *Signal Processing*, 82(1):117–120, 2002.

[34] Anthony Karel Seda. On the integration of connectionist and logic-based systems. In T. Hurley, M. Mac an Airchinnigh, M. Schellekens, A.K. Seda, and G. Strong, editors, *Proceedings of MFCSIT2004, Trinity College Dublin, July 2004*, Electronic Notes in Theoretical Computer Science, pages 1–24. Elsevier, 2005.

[35] Anthony Karel Seda and Maire Lane. On approximation in the integration of connectionist and logic-based systems. In *Proceedings of the Third International Conference on Information (Information'04)*, pages 297–300, Tokyo, November 2005. International Information Institute.

[36] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Džeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang. The MONK's problems: A performance comparison of different learning algorithms. Technical Report CS-91-197, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1991.

[37] A. B. Tickle, F. Maire, G. Bologna, R. Andrews, and J. Diederich. Lessons from past, current issues, and future research directions in extracting the knowledge embedded in artificial neural networks. *Hybrid Neural Systems*, pages 226–239, 1998.

[38] Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1–2):119–165, 1994.