Extracting Reduced Logic Programs from Artificial Neural Networks

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

Abstract

Artificial neural networks can be trained to perform excellently in many application areas. While 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 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; 8; 14; 18; 19] 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 behaviour 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 behaviour 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 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; 4; 5; 6; 11; 12], 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 [10]. While our broader 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 not been answered yet within the avail-

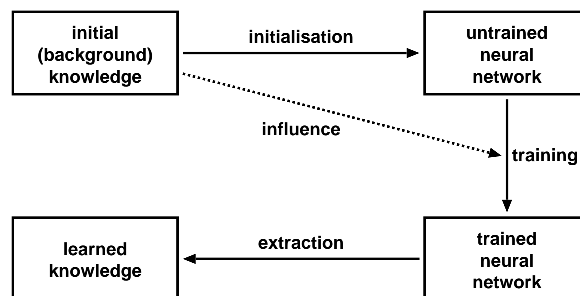

Figure 1: Neural-symbolic learning cycle

*Sebastian Bader is supported by the GK334 of the German Research Foundation (DFG).

†Pascal Hitzler is supported by the German Federal Ministry of Education and Research under the SmartWeb project, and by the European Commission under contract IST-2003-506826 SEKT.

able 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 will show 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 will also clarify formally what we mean by *redundancy* in this case. In the presence of negation, i.e. for *normal* logic programs, unique representations cannot be obtained in general, but we will investigate methods and present algorithms for removing redundancies.

The structure of the paper is as follows. After some preliminaries reviewed in Sections 2 and 3, we will present our main result on the extraction of a unique irredundant definite logic program in Section 4. How to remove redundancies in normal logic programs is discussed in Section 5, while a corresponding algorithm is presented in Section 6.

2 Logic Programs

We first introduce some standard notation for logic programs, roughly following [16]. 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 [16]. 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 $\mathcal{P}$:*

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

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

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

1. *There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in $\mathcal{P}$, such that C_1 subsumes C_2.*
2. *A predicate symbol does not appear more than once in any clause body.*
3. *No clause body contains a predicate and its negation.*

Condition 3 is actually redundant, as it is covered by condition 2. Nevertheless, we have chosen to state it seperately as this form of presentation appears to be more intuitive. Humans usually write reduced logic programs.

Using Definition 2.2, we can define a naive algorithm 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. The following result is obvious.

Proposition 2.3. *If Q is a reduced version of the propositional logic program $\mathcal{P}$, then $T_{\mathcal{P}} = T_{Q}$.*

3 Neural-Symbolic Integration

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 refer to [7] for background on artificial neural networks.

For our purposes, 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 behaviour 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 [12; 13], 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 behaviour of the network matches the $T_\mathcal{P}$-operator. The basic idea is depicted in Figure 3, and an example-run of the network is shown in Figure 4. The algorithm was generalized to sigmoidal activation functions in [10], thus enabling the use of powerful learning algorithms based on backpropagation [7].

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 $f : I_\mathcal{P} \to I_\mathcal{P}$ 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 in a way such that $\mathcal{P}$ is as simple as possible, i.e. minimal respectively reduced.

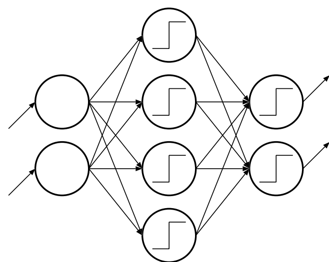

Figure 2: A simple 3-layer feed forward neural network with threshold activation function.

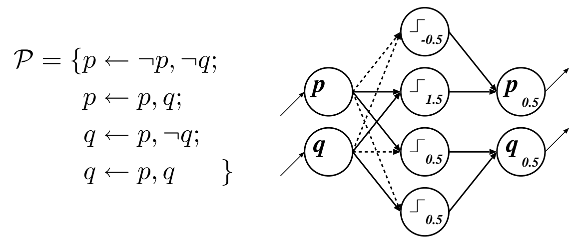

Figure 3: The 3-layer network constructed to implement the $T_\mathcal{P}$-operator of the given program $\mathcal{P}$. Connections with weight 1 are depicted solid, those with weight -1 are dashed. The numbers denote the thresholds of the units.

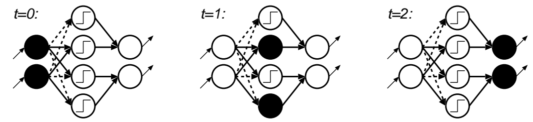

Figure 4: A run of the network depicted in Figure 3 for the interpretation $I = \{p, q\}$. A unit is depicted in black, if its activation is 1. At time $t = 0$ the corresponding units in the input layer are activated to 1. This activation is propagated to the hidden layer and results in two active units there. Finally, it reaches the output layer, i.e. $T_\mathcal{P}(I) = \{p, q\}$.

We start with naive extraction by *"Full Exploration"*, detailed in Algorithms 1 and 2, for definite respectively normal logic program extraction. We will see later that the extraction of definite programs is easier and theoretically more satisfactory. However, negation is perceived as a highly desirable feature because in general it allows to express knowledge more naturally. The target function itself does not limit the choice, so which approach will be chosen for a problem at hand will depend on the application domain. We give an example for full exploration in the normal case.

Example 1. *Let $I_\mathcal{P} = \{p, q\}$ and the mapping f be obtained by full exploration of the network shown in Figure 3. Using Algorithm 2, we obtain program $\mathcal{P}$ again, and $T_\mathcal{P} = f$ holds.*

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

Using Algorithm 2, the following result is easily obtained.

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

Note that programs obtained using Algorithms 1 or 2 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 1 or 2 and refine it. This general idea was first described in [13], but not spelled out using an algorithm. On

Algorithm 1 Full Exploration — Definite

Let f be a mapping from $I_{\mathcal{P}}$ to $I_{\mathcal{P}}$. Initialize $\mathcal{P} = \emptyset$. For every interpretation $I = \{r_1, \ldots, r_a\} \in I_{\mathcal{P}}$ and each element $h \in f(I)$ add a clause $h \leftarrow r_1, \ldots, r_a$ to $\mathcal{P}$. Return $\mathcal{P}$ as result.

Algorithm 2 Full Exploration — Normal

Let f be a mapping from $I_{\mathcal{P}}$ to $I_{\mathcal{P}}$. Initialize $\mathcal{P} = \emptyset$. For every interpretation $I = \{r_1, \ldots, r_a\} \in I_{\mathcal{P}}$, we have $B_{\mathcal{P}} \setminus I = \{s_1, \ldots, s_b\}$. For each element $h \in f(I)$ add a clause $h \leftarrow r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ to $\mathcal{P}$. Return $\mathcal{P}$ as result.

the other hand, we can build a program from scratch. Both possibilities will be pursued in the sequel.

4 Extracting Reduced Defi nite 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 also show that the notion of minimal program coincides with that of reduced program, thus serving both intuitions at the same time. Algorithm 3 satisfies our requirements, as we will see shortly.

Proposition 4.1. *Let $T_{\mathcal{P}}$ be a definite consequence operator and $\mathcal{Q}$ be the result of Algorithm 3, obtained for $f = T_{\mathcal{P}}$. Then $T_{\mathcal{P}} = T_{\mathcal{Q}}$.*

Proof. We have to show $T_{\mathcal{P}}(I) = T_{\mathcal{Q}}(I)$ for an arbitrary interpretation $I = \{p_1, \ldots, p_n\}$.

For $T_{\mathcal{P}}(I) \subseteq T_{\mathcal{Q}}(I)$ we have to show that $q \in T_{\mathcal{Q}}(I)$ holds if $q \in T_{\mathcal{P}}(I)$. Assume we have a predicate q in $T_{\mathcal{P}}(I)$. 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 $q \in T_{\mathcal{Q}}(I)$.
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 $T_{\mathcal{Q}}(I) \subseteq T_{\mathcal{P}}(I)$. As in the other direction, we now have a predicate q in $T_{\mathcal{Q}}(I)$ and have to show that it is also 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 3 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 2.2 there are two possible reasons for this:

Case 1: A predicate symbol appears more than once in the body of a clause. This is impossible, because the algorithm

Algorithm 3 Extracting a Reduced Definite Program

Let $f : I_{\mathcal{P}} \to I_{\mathcal{P}}$ be a given mapping, as obtained e.g. from a neural network, and consider $I_{\mathcal{P}}$ to be totally ordered in some way such that I is before K in the ordering if $|I| < |K|$. Let $\mathcal{Q}$ be an initially empty program.

For all interpretations $I \in I_{\mathcal{P}}$, in sequence of the assumed ordering, do the following:

- Let $I = \{p_1, \ldots, p_n\}$. For every $q \in f(I)$, check whether a clause $q \leftarrow q_1, \ldots, q_m$ with $\{q_1, \ldots, q_m\} \subseteq I$ is already in $\mathcal{Q}$. If not, then add the clause $q \leftarrow p_1, \ldots, p_n$ to $\mathcal{Q}$.

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

does not generate such clauses (sets do not contain elements twice).

Case 2: There are two different clauses C_1 and C_2 in $\mathcal{Q}$, such that C_1 subsumes C_2. Let C_1 be $h \leftarrow p_1, \ldots, p_a$ and C_2 be $h \leftarrow q_1, \ldots, q_b$ with $\{p_1, \ldots, p_a\} \subseteq \{q_1, \ldots, q_b\}$. As abbreviations we use $I = \{p_1, \ldots, p_a\}$ and $J = \{q_1, \ldots, q_b\}$. Because of case 1 we know $|I| = a$ and $|J| = b$ (all elements in the body of a clause are different). Thus we have $|I| < |J|$, because C_1 and C_2 are not equal. 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 (otherwise the body respectively head of the clause would be different). Later the case J and h was treated. The algorithm checks for clauses $h \leftarrow r_1, \ldots, r_m$ with $\{r_1, \ldots, r_m\} \subseteq J$. C_1 is such a clause, because $I \subseteq J$, so C_2 is not added to $\mathcal{Q}$. Because (by the same argument as above) C_2 can only be generated through J and h, C_2 cannot be a clause in $\mathcal{Q}$, which is a contradiction and completes the proof. □

Propositions 4.1 and 4.2 have shown that the output of the extraction algorithm is in fact a reduced definite program, which has the desired operator. We proceed to show that the obtained reduced program is unique. The following, together with Corollary 4.4, is the main theoretical result 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 3 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

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

C_3 subsumes C_1, because it has the same head and $\{r_1, \dots, r_o\} \subseteq \{q_1, \dots, q_n\} \subseteq \{p_1, \dots, p_m\}$. 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 $|\{p_1, \dots, q_m\}| = m$ and $|\{q_1, \dots, q_n\}| = n$, because P_1 and P_2 are reduced, i.e. no predicate appears more than once in any clause body. So we have $\{q_1, \dots, q_n\} \subset \{p_1, \dots, p_m\}$. 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 is a contradiction to P_1 being reduced. □

This shows that each algorithm extracting reduced definite programs from a neural network must return the same result as Algorithm 3. We can now also obtain that the notion of reduced program coincides with that of minimal program, which shows that Algorithm 3 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 a 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, so assume it is not, i.e $\mathcal{Q}$ can be reduced. The resulting program $\mathcal{Q}_{red}$ is definite, by Definition 2.2 obviously smaller than before the reduction, and has operator $T_{\mathcal{P}} = T_{\mathcal{Q}}$. 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 $\mathcal{Q}$, $\mathcal{P}$ is also smaller than $\mathcal{Q}$. □

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 behaviour 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 2 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.

Example 2. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be defined as follows:*

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

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 2.2.

Example 2 shows that the notion of reduction in terms of Definition 2.2 is insufficient for normal logic programs. *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). 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. But it raises the question: *Is there always a unique minimal (i.e. least) program for any given target function?* The answer is negative, as the following example shows.

Example 3. *The following programs are equivalent.*

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

A full search easily reveals that the given two programs are minimal. We skip the details, which can be found in [15].

Example 3 shows that an analogy to Corollary 4.4 does not hold for normal programs. This means that we can at best hope to extract minimal normal programs from neural networks, but in general not a least program. The complexity of this task is as yet unknown, as is an optimal extraction algorithm, but we will later be able to discuss a refinement of the naive algorithm given earlier.

For the moment, we will shortly discuss possibilities for refining the set obtained by Algorithm 2. We start with two examples.

Example 4. *Let $\mathcal{P}_1$ be defined as introduced in Example 1:*

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

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 both by $q \leftarrow p$. The resulting program is shown as $\mathcal{P}_2$.

Another case is given by the setting in Example 2, where a similar situation occurs. By generalizing from these examples, we arrive at the following notion.

Definition 5.1. *An α-reduced program $\mathcal{P}$ is a program with the following properties.*

1. *$\mathcal{P}$ is reduced.*
2. *There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in $\mathcal{P}$, where C_1 is of the form $p \leftarrow q, r_1, \dots, r_a, \neg s_1, \dots, \neg s_b$ and C_2 is of the form $p \leftarrow \neg q, t_1, \dots, t_c, \neg u_1, \dots, \neg u_d$, where $\{r_1, \dots, r_a\} \subseteq \{t_1, \dots, t_c\}$ and $\{s_1, \dots, s_b\} \subseteq \{u_1, \dots, u_d\}$.*
3. *There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in $\mathcal{P}$, where C_1 is of the form $p \leftarrow \neg q, r_1, \dots, r_a, \neg s_1, \dots, \neg s_b$ and C_2 is of the form $p \leftarrow q, t_1, \dots, t_c, \neg u_1, \dots, \neg u_d$, where $\{r_1, \dots, r_a\} \subseteq \{t_1, \dots, t_c\}$ and $\{s_1, \dots, s_b\} \subseteq \{u_1, \dots, u_d\}$.*

Both Example 2 and 4 show logic programs and their α-reduced versions. The following result and corresponding Algorithm 4 can be obtained, for details we refer to [15].

Algorithm 4 Constructing an α-reduced program

For an arbitrary propositional logic program $\mathcal{P}$ perform the following reduction steps as long as possible:

1. If there are two clauses C_1 and C_2 such that point 2 of Definition 5.1 is fulfilled, then remove $\neg q$ in the body of C_2.
2. If there are two clauses C_1 and C_2 such that point 3 of Definition 5.1 is fulfilled, then remove q in the body of C_2.
3. If there are clauses C_1 and C_2 with $C_1 \neq C_2$ in $\mathcal{P}$ and C_1 subsumes C_2, then remove C_2.
4. If a literal appears twice in the body of a clause, then remove one occurrence.
5. If a predicate and its negation appear in the body of a clause, then remove this clause.

Proposition 5.2. *Let $\mathcal{P}$ be a logic program. If $\mathcal{Q}$ is the result of Algorithm 4 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. *The following two programs are equivalent. $\mathcal{P}_2$ is as in Example 3.*

$$\begin{array}{ll} \mathcal{P}_2 = \{p \leftarrow \neg p, \neg r; & \mathcal{P}_3 = \{p \leftarrow \neg p, \neg r; \\ \quad p \leftarrow p, r; & \quad p \leftarrow p, r; \\ \quad p \leftarrow q, r \quad \} & \quad p \leftarrow q, r; \\ & \quad p \leftarrow \neg p, q \quad \} \end{array}$$

Even though both programs are α-reduced, P_3 is larger than P_2. Note also that P_3 can be transformed to P_2 by removing a redundant clause. However, this cannot be done by α-reduction.

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 [17]. Such a line of action was spelled out in [15]. The resulting algorithms did yield further refined programs at the cost of lower efficiency, but no satisfactory algorithms for obtaining minimal programs.

6 A Greedy Extraction Algorithm

We present another extraction algorithm for normal programs, which is closer in spirit to Algorithm 3 in that it incrementally builds a program. For this purpose we introduce the notion of *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.

The following example illustrates the intuition.

Example 6. *We will use the operator of the programs given in Example 3:*

$$\begin{array}{llll} T_{\mathcal{P}} = \{ & \emptyset \mapsto \{p\} & & \{p,q\} \mapsto \emptyset \\ & \{p\} \mapsto \emptyset & & \{p,r\} \mapsto \{p\} \\ & \{q\} \mapsto \{p\} & & \{q,r\} \mapsto \{p\} \\ & \{r\} \mapsto \emptyset & & \{p,q,r\} \mapsto \{p\}\} \end{array}$$

The 3 atoms p, q, r are being used, so there would be 27 different possible clauses bodies, as shown in Table 1. The clause $p \leftarrow p$, for example, is not correct, since we have $p \notin T_{\mathcal{P}}(\{p\})$. Hence the body p is not allowed.

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

Definition 6.1. *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:*

- *For every interpretation $I \subseteq B_{\mathcal{P}}$ with $I \models B$ we have $h \in T_{\mathcal{P}}(I)$.*
- *There is no allowed body $B' = r_1, \ldots, r_c, \neg t_1, \ldots, \neg t_d$ for h and $T_{\mathcal{P}}$ with $B' \neq B$ such that $\{r_1, \ldots, r_c\} \subseteq \{p_1, \ldots, p_a\}$ and $\{t_1, \ldots, t_d\} \subseteq \{q_1, \ldots, q_b\}$.*

As given in Definition 6.1, there are two reason for a clause body B not to be allowed. First, the resulting clause could be wrong, as discussed in Example 6. Secondly, there could be a smaller allowed body B', such that $h \leftarrow B'$ subsumes $h \leftarrow B$.

Example 6 (continued). *Table 1 shows all possible clause bodies for $B_{\mathcal{P}} = \{p, q, r\}$ on the left side. The right side shows either "OK", if the body is allowed, or gives the reason why it is not allowed.*

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 6.2. *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*

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

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 $T^q_{\mathcal{P}}$ the *restricted* consequence operator for q and set $T^q_{\mathcal{P}}(I) = \{q\}$ if $q \in T_{\mathcal{P}}(I)$, and $T^q_{\mathcal{P}}(I) = \emptyset$ otherwise. Algorithm 5 gives the details of the resulting procedure and is illustrated in Example 7.

clause body	evaluation
empty	False, $p \notin T_{\mathcal{P}}(\{p\})$.
p	False, because $p \notin T_{\mathcal{P}}(\{p\})$.
q	False, because $p \notin T_{\mathcal{P}}(\{p,q\})$.
r	False, because $p \notin T_{\mathcal{P}}(\{r\})$.
$\neg p$	False, because $p \notin T_{\mathcal{P}}(\{r\})$.
$\neg q$	False, because $p \notin T_{\mathcal{P}}(\{p\})$.
$\neg r$	False, because $p \notin T_{\mathcal{P}}(\{p\})$.
p, q	False, because $p \notin T_{\mathcal{P}}(\{p,q\})$.
p, r	OK.
q, r	OK.
$p, \neg q$	False, because $p \notin T_{\mathcal{P}}(\{p\})$.
$p, \neg r$	False, because $p \notin T_{\mathcal{P}}(\{p\})$.
$q, \neg p$	OK.
$q, \neg r$	False, because $p \notin T_{\mathcal{P}}(\{p,q\})$.
$r, \neg p$	False, because $p \notin T_{\mathcal{P}}(\{r\})$.
$r, \neg q$	False, because $p \notin T_{\mathcal{P}}(\{r\})$.
$\neg p, \neg q$	False, because $p \notin T_{\mathcal{P}}\{r\})$.
$\neg p, \neg r$	OK.
$\neg q, \neg r$	False, because $p \notin T_{\mathcal{P}}(\{p\})$.
p, q, r	Not considered, because p, r is smaller.
$p, q, \neg r$	False, because $p \notin T_{\mathcal{P}}(\{p,q\})$.
$p, \neg q, r$	Not considered, because p, r is smaller.
$\neg p, q, r$	Not considered, because q, r is smaller.
$p, \neg q, \neg r$	False, because $p \notin T_{\mathcal{P}}(\{p\})$.
$\neg p, q, \neg r$	Not considered, because $\neg p, q$ is smaller.
$\neg p, \neg q, r$	False, because $p \notin T_{\mathcal{P}}(\{r\})$.
$\neg p, \neg q, \neg r$	Not considered, because $\neg p, \neg r$ is smaller.

Table 1: Allowed clause bodies for $T_{\mathcal{P}}$ from Example 6.

Example 7. *Let $T_{\mathcal{P}}$ be given as follows:*

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

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

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

Tables 2 and 3 show two example runs of the algorithm. 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.) As an example the score for $p, q, \neg s$ in the first step of the first run is 2, because $p \in T_{\mathcal{P}}(p,q)$ and $p \in T_{\mathcal{P}}(p,q,r)$. It goes down to 1 in the second step, because we have $\mathcal{Q} = \{p, r\}$ and therefore $p \in T_{\mathcal{Q}}(p,q,r)$ at this point. Intuitively this means that we would only gain one additional interpretation by adding $p \leftarrow p, q, \neg s$.

Algorithm 5 Greedy Extraction Algorithm

Let $T_{\mathcal{P}}$ and $B_P = \{q_1, \ldots, q_m\}$ be the input of the algorithm.
Initialize $\mathcal{Q} = \emptyset$.
For each predicate $q_i \in B_{\mathcal{P}}$:

1. construct the set S_i of allowed clause bodies for q_i
2. initialize: $\mathcal{Q}_i = \emptyset$
3. repeat until $T_{\mathcal{Q}_i} = T_{\mathcal{P}}^{q_i}$:
   (a) Determine a clause C of the form $h \leftarrow B$ with $B \in S_i$ with the highest score with respect to $\mathcal{Q}_i$.
   (b) If several clauses have the highest score, then choose one with the smallest number of literals.
   (c) $\mathcal{Q}_i = \mathcal{Q}_i \cup \{C\}$
4. $\mathcal{Q} = \mathcal{Q} \cup \mathcal{Q}_i$

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

---

clause body	1.	2.	3.	4.	5.	6.
p, r	**4**					
$\neg p, \neg r, \neg s$	2	2	**1**			
$q, \neg p, \neg r$	2	**2**				
$q, \neg r, \neg s$	2	2	1	**1**		
$p, q, \neg s$	2	1	1	1	0	0
$p, s, \neg q$	2	1	1	1	**1**	
$q, s, \neg p$	2	2	1	1	1	**1**
q, r, s	2	1	1	1	1	1

$$\mathcal{P}_1 = \{p \leftarrow p, r; \quad p \leftarrow q, \neg p, \neg r; \quad p \leftarrow \neg p, \neg r, \neg s; \quad p \leftarrow q, \neg r, \neg s; \quad p \leftarrow p, s, \neg q; \quad p \leftarrow q, s, \neg p \quad \}$$

Table 2: Example run 1 and the resulting program.

Example 7 is constructed in such a way that 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 as 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). As for performance, the use of allowed clause bodies in this case made it possible to reduce checking from 27 to 4 clauses.

Let us finally mention how to modify Algorithm 5 in order

clause body	1.	2.	3.	4.	5.
p, r	**4**				
$\neg p, \neg r, \neg s$	2	**2**			
$q, \neg p, \neg r$	2	2	1	0	0
$q, \neg r, \neg s$	2	2	1	**1**	
$p, q, \neg s$	2	1	1	1	0
$p, s, \neg q$	2	1	1	1	**1**
$q, s, \neg p$	2	2	**2**		
q, r, s	2	1	1	0	0

$$\mathcal{P}_2 = \{p \leftarrow p, r; \quad p \leftarrow \neg p, \neg r, \neg s; \quad p \leftarrow q, s, \neg p; \quad p \leftarrow q, \neg r, \neg s; \quad p \leftarrow p, s, \neg q \quad \}$$

Table 3: Example run 2 and the resulting program.

Algorithm 6 Intelligent Program Search

Let $T_{\mathcal{P}}$ and $B_{\mathcal{P}} = \{q_1, \ldots, q_m\}$ be the input of the algorithm.
Initialize: $Q = \emptyset$.
For each predicate $q_i \in B_{\mathcal{P}}$:

1. construct the set S_i of allowed clause bodies for q_i
2. initialize: $n_i = 0$
3. Search all programs with size equal to n_i until a program $\mathcal{Q}_i$ with $T_{\mathcal{P}}^{q_i} = T_{\mathcal{Q}_i}$ is found. if no such program is found then increment n_i and repeat step 3.
4. $\mathcal{Q} = \mathcal{Q} \cup \mathcal{Q}_i$

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. See Algorithm 6.

7 Conclusions

We presented algorithms to extract 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 minimal program with the desired operator; and it was formally shown that such a minimal 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 the automatic refinement of logic programs, obtained by global extraction methods as in [9; 13]. We have thus addressed and answered fundamental (and obvious) open questions. 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 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 and P. Hitzler. Logic programs, iterated function systems, and recurrent radial basis function networks. *Journal of Applied Logic*, 2(3):273–300, 2004.

[4] S. Bader, P. Hitzler, and A. S. d'Avila Garcez. Computing first-order logic programs by fibring artificial neural network. In *Proceedings of the 18th International FLAIRS Conference, Clearwater Beach, Florida, May 2005*, 2005. To appear.

[5] 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.

[6] S. Bader, P. Hitzler, and A. Witzel. Integrating first-order logic programs and connectionist systems — a constructive approach. In *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK*, 2005. To appear.

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

[8] 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.

[9] 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.

[10] 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.

[11] P. Hitzler, S. Bader, and A. Garcez. Ontology learning as a use-case for neural-symbolic intergration. In *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK*, 2005. To appear.

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

[13] 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.

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

[15] J. Lehmann. Extracting logic programs from artificial neural networks. Belegarbeit, Fakultät Informatik, Technische Universität Dresden, February 2005.

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

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

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

[19] 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.