Rule Extraction From Artificial Neural Networks Under Background Knowledge

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Applied Mathemtics

by

Maryam Labaf
B.S, Esfahan University, 2007

2017
Wright State University

Wright State University
SCHOOL OF GRADUATE STUDIES

July 2017

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Maryam Labaf ENTITLED Rule Extraction From Artificial Neural Networks Under Background Knowledge BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Applied Mathmatics.

Anthony B. Evans, Ph.D.
Thesis Director

Pascal Hitzler, Ph.D.
Thesis Co-Director

Ayse Sahin, Ph.D.
Chair, Department of
Mathemtics and
Statistics

Committee on
Final Examination

Pascal Hitzler, Ph.D.

Anthony B.Evans, Ph.D.

Ayse Sahin, Ph.D.

ABSTRACT

Labaf, Maryam. M.S, Department of Mathematics and Statistics, Wright State University, 2017. *Rule Extraction From Artificial Neural Networks Under Background Knowledge.*

It is well-known that the input-output behavior of a neural network can be recast in terms of a set of propositional rules, and under certain weak preconditions this is also always possible with positive (or definite) rules. Furthermore, in this case there is in fact a unique minimal (technically, *reduced*) set of such rules which perfectly captures the input-output mapping.

In this dissertation, we investigate to what extent these results and corresponding rule extraction algorithms can be lifted to take additional background knowledge into account. It turns out that uniqueness of the solution can then no longer be guaranteed. However, the background knowledge often makes it possible to extract simpler, and thus more easily understandable, rulesets which still perfectly capture the input-output mapping.

Contents

List of Figures

List of Algorithms

Acknowledgment

I would first like to take this opportunity to extend my special thanks to my thesis advisor, Professor Anthony B. Evans of the Mathematics Department at Wright State University, for his unconditional help and support during my master thesis.

Also, I would like to express my sincere gratitude to my co–advisor, Professor Pascal Hitzler of the Computer Science Department at Wright State University, who has mentored me through my master thesis with his expertise, care, and patience. He gave me the opportunity to work in a different area of my background and taught me how to write a scientific article. He commented on my weekly reports and guided me in the right direction whenever I needed it. Furthermore, thanks for his financial support through my final semester at Wright State University.

I am extremely fortunate to have these two expert advisors from two different programs. Their different view points from their respective academic backgrounds gave me insight to conduct my research. I really appreciate their time and effort reading and commenting on my work. This dissertation would not have been possible without their help. My confidence to further pursue my studies in a doctoral program is mainly inspired by their passion.

I would like to thank Professor Ayse Sahin for serving on my committee and providing valuable insights to my research. Additionally, I appreciate the Mathematics Department's help in providing financial assistance through teaching assistantships.

Finally, I would like to extend my most profound gratitude to my husband and my family for providing me with unfailing support and continuous encouragement throughout my years of study.

Introduction and Motivation

Artificial neural networks (ANNs) have the ability to learn complex nonlinear input-output relationships. They are also a great way to learn decision boundary problems to distinguish the positive and negative examples. ANNs were widely used in 80's and early 90's. Their popularity diminished in late 90's when machine learning move towards the convex optimization methods. Throughout the 80's and 90's there was not enough computational power in order to train the highly nonlinear non-convex and very complicated computations that neural networks need. But nowadays, thanks to technology advancement, neural networks is one of the state–of–the–art techniques for many applications.

The efficiency of computing models of ANN to solve complex problems in artificial intelligence is definite. ANN is an information processing paradigm, made of a large number of interconnected processing elements, which was inspired by the way biological nervous system works. During the last decades, applying neural networks to solve a variety of problems, such as pattern recognition, medical diagnosis, stuck markets, data classification, prediction, and function approximation have been the focus of a large number of researchers and practitioners. For pattern recognition, we could use feedforward networks (see 2.4). The main reasons for using neural networks to solve pattern recognition problems is low dependence on domain-specific knowledge and availability of efficient learning algorithms for practitioners.

One of the main concerns of data classification and prediction research is deriving understandable rules from data analysis, e.g., discover the hidden rules and structures from an existing large amount of data. Trained neural networks are *black boxes*; despite the application of neural network as a predictive models, sometimes it cannot be classified as a satisfiable data mining tool for pattern discovery. Furthermore, the knowledge learned by neural networks is hard to understand. Also, we cannot find an explicit explanation of the mechanism inside the trained neural networks due to a large amount of connections [22, 23]. In most of the critical application areas, extracting rules from trained neural networks can logically justify the knowledge embedded in the network. Additionally, understanding the internal logic of the system gives us a better insight into a problem. To address all of these limitations, various approaches have been developed to extract useful knowledge from trained neural networks.

There needs to be a three-step process to use neural networks for the refinement of existing knowledge: (1) inserting knowledge into a neural network; (2) refining the network, and; (3) extracting knowledge from the refined network [19]. For the last method we are looking for an efficient method to extract symbolic rules from trained neural networks. The study of rule extraction has two main goals – attain an exact and comprehensive learning system, and understand the working mechanism of the neural network [22]. Propositional logic program is one of the eminent methods which mainly focus on symbolic integration. More precisely, the input-output values are identical with truth values for propositional variables and the behavior of the network can be described in term of logical rules.

In this dissertation, we investigate rule extraction from trained neural networks under the assumption that there is additional background knowledge which can be connected to network node activations. This background knowledge will make it possible to formulate the simpler rulesets which still explain the input-output functions of the networks, if the background knowledge is also taken into account. In recent years there has been a very considerable growth in the availability of the structured data on the World Wide Web, i.e., it becomes easier and easier to actually find such structured knowledge for all different kinds of application domains. This being the case, it is, among other things, a result of recent developments in the field of *Semantic Web* [6, 11], which is concerned with data sharing, discovery, integration, and reuse, and where corresponding standards, methods and tools are being developed. For example., structured data in the form of *knowledge graphs*, usually encoded using the W3C standards RDF [5] and OWL [10], has been made available in ever increasing quantities for over 10 years [7, 17]. Other large-scale datasets include Wikidata [20] and data coming from the schema.org [9] effort which is driven by major Web search engine providers.

In the following chapters , we will first review and introduce ANN and symbolic representation of neural networks as a connectionist system. After some preliminaries are reviewed in chapter 3, we will give an overview of some of the theoretical results on the rule extraction methods of propositional logical program, which prepare the main idea of our work in chapter 6. The main focus of this chapter 4 is explaining Lehnamm 's result [16] of extracting a unique irredundant definite logic program. In chapter 5, we introduce recurrent network and core methods. Chapter 6 focuses on our results regarding rulesets properties for definite propositional logic program under background knowledge together with corresponding algorithms. We will show under background knowledge the uniqueness of the solutions can no longer be guaranteed. A version of this chapter has been published in *International Workshop on Neural-Symbolic Learning and Reasoning (NESY'17)* [15]. The conclusion and final work are drawn in the last chapter.

Artificial Neural Networks

One of the research areas in Artificial Intelligence is inspired from the neural networks of human nervous system. How is the information represented in the brain? If we think of the brain as a neural network, with firing and synchronization of nodes, synapses, and transmitters, how is the knowledge encoded there?

The idea of ANNs is based on how the neurons in the human brain process information. The brain is composed of tens of billions of neurons, which send information through their axons and receive information through their dendrites. Information takes the form of electrical impulses, which quickly travel through this neural network. A neuron processes the information and can decide whether or not to transmit the information to other neurons. More precisely, the dendrites of the neuron are considered as inputs and, depending on what input the neuron receives, the neuron cell body performs computations and determines whether or not to send an electrical pulse. The output wires are connected to other neurons and this is how the human brain works.

Similarly, ANNs are composed of multiple nodes, where a node is equivalent to a neuron and the multiple nodes are equivalent to a neural network. These nodes are connected by links, which are equivalent to the axons and dendrites, and interact with other nodes via the links. The nodes can then receive information and perform simple operations on the information. The result of these operations can be passed to other nodes in the network. The output at each node is called its activation or node value. Each link is associated with a weight that controls the signal between neurons. ANNs are capable of learning, which

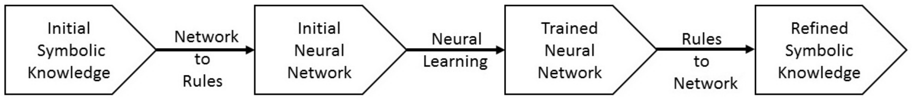


Figure 2.1: Three-Link Chain of Rule Refinement

takes place by altering weight values. Adjusting the weights can be performed when the network does not generate a desired output or error. In order to improve the consequent results in the system, we should alter the weights.

2.1 Neural Symbolic Representation

We could use a three-link chain of neural networks in order to revise and correct the symbolic knowledge. See Figure 2.1. First, we should insert knowledge into the neural network, which becomes a knowledge-based neural networks (KNN). This step changes the rule from symbolic to neutrally based. Second, train KNN using any standard neural network algorithm, e.g., backpropagation or any other weight optimized method of feedforward neural networks. At the end, we need to extract the rules from the trained KNNs [19].

The main question is how symbolic knowledge is represented within the connectionist system in a suitable way. It is not easy to identify one or a small number of the nodes whose activations contain and process a certain symbolic piece of knowledge. In order to insert knowledge into the neural network, we could use the connectionist weights of the network by applying the knowledge base representation of rules as propositional horn clauses [19]. Specifically, this method deals with symbolic knowledge within connectionist system by using logical representation. The landmark paper by McCulloch and Pitts [21] provides the fundamental insights on how symbolic knowledge in term of propositional logic can represent neural networks. By using the connectionist representation of symbolic knowledge, we could extract symbolic knowledge from trained ANNs. McCulloch-Pitts in 1943 simulated neurons with binary activation functions and made a propositional connectives model, making the network equivalent to finite automata. In order to apply a logical representation of neural networks, the input-output values are identified with truth values for propositional variables and the behavior of the network can be described in terms of logical rules. McCulloch and Pitts [21] used the ”all-or-none” character of nervous activity and constructed a very simple mathematical model of neural activity. Each neuron is simulated as one node such that the nodes (neurons) are connected to each other using the weighted connections, with each node having an activation threshold. In this mathematical simulation, a group of input nodes needs to exceed the threshold to fire. If it does not exceed the threshold, the node does not fire. Neural events and the connections among them can be treated by means of propositional logic. In this model, the behavior of even the most complicated nets can be interpreted in the notation of the symbolic logic of propositions by encoding the network to logical connections – disjunction $\vee$, conjunction $\wedge$, negation $\neg$, implication $\rightarrow$, and equivalence $\equiv$. In this simple interpretation, we could process all logical inputs and produce some outputs. Actually, it is possible to correlate the physiological relations existing among nervous activities with relations among the propositions; and the usage of the representation depends upon the identity of these relations with those of the logic propositions. Due to the time dependency of the activities in the nervous system and the existence of circular path between nodes, McCulloch and Pitts [21] applied some physical primitive assumptions to complete the theory: (1) the nets should not be cyclic, i.e. no circle path; (2) the activity of the neuron is an ”on-off” process; (3) at any time a certain number of the neurons must be fired and they are independent of the previous action and position of the neurons, and; (4) the general structure of the system is time independent.

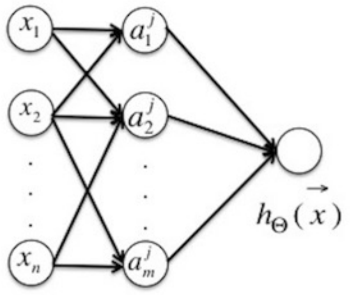


Figure 2.2: Three –Layer Neural Network

2.2 Connectionist System

An ANN, which is also called *connectionist system*, consists of a finite set $\mathcal{U}$ of units (input, hidden and output) and a set $\mathcal{W} \subseteq \mathcal{U} \times \mathcal{U}$ of weighted directed connections. Each connection labeled by a weight $w \in \mathbb{R}$ such that if there is a connection from unit u_i to u_j, then w_{ji} is its associated weight. A unit is specified by: (1) an input vector $\vec{i} = (i_1, i_2, ..., i_m), i_j \in \mathbb{R}, 1 \leq j \leq m$; (2) an activation function Φ mapping $\vec{i}$ to a potential $p \in \mathbb{R}$ such that $p = \sum_{r=1}^{r=m} i_r.w_r$ or $p = \sum_{r=1}^{r=m}(i_r - w_r)^2$, and; (3) an output function Ψ mapping p to an output value $v \in \mathbb{R}$. If there is a connection from u_i to u_j, then $w_{jk}v_i$ is the input received by u_j from u_i along this connection at time t. Through learning the network the potential and output value of a unit are simultaneously recomputed or updated. McCulloch and Pitts [21] simulated the activities of nervous systems by logical calculus. Their network follows the three –layer connectionist system of input, output, and hidden layers. In the following, we show how this neural network connectionist system works and how the logical model represented by McCulloch and Pitts [21] works. We utilize the McCulloch–Pitts model to train a neural network to learn the logic functions ”AND”, ”OR”, and ”NOR”. Other logic functions can be extracted by applying the same method.

2.3 Neural Logical Model

As mentioned earlier, the neuron model is a simple logistic unit. Each neuron has input units x_i where $i \geq 1$. After doing computations there is an output function $h_w(\vec{x})$, where $\vec{x}$ is a vector of input units, and w is a vector of parameters. Each neuron takes the inner product between $\vec{x}$ and w and applies the logistic (sigmoid) activation function to find the simulation of the neuron. We compute the total weighted inputs and calculate the output using the logistic sigmoid activation function. In Figure 2.2, there is a three–layer neural network with a layer of input units x_i, $a_i^{(j)}$ activation of unit x_i in layer j, w is a matrix of weights and $h_w(\vec{x})$ is a logistic regression function that applies to the features $a_i^{(j)}$ in hidden layer j. Each layer has some features which represent the data. Each node takes an inner product of $\vec{x}$ with some of the features of w matrix. Applying the sigmoid function, which defined in the following, to each linear combination of input and output unit values computed in the previous layer. If the network has s_j units in layer j, and s_{j+1} units in layer $j+1$, then w will be of dimension $s_{(j+1)} \times (s_j + 1)$.

Definition 1. ***Sigmoid Function:*** *A sigmoid function, also known as a logistic function, is defined by the formula* $(t) = \frac{1}{1+e^{-\beta t}}$, *where* β *is a slope parameter. In order to use the sigmoidal function as an activation function,* $\beta t = (p_k + \theta_k)$, *where* p_k *and* θ_k *are the potential and value of the of the unit* k, *respectively.*

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } w = \begin{bmatrix} \vec{w}_0 \\ \vec{w}_1 \\ \vdots \\ \vec{w}_m \end{bmatrix} \tag{2.1}$$

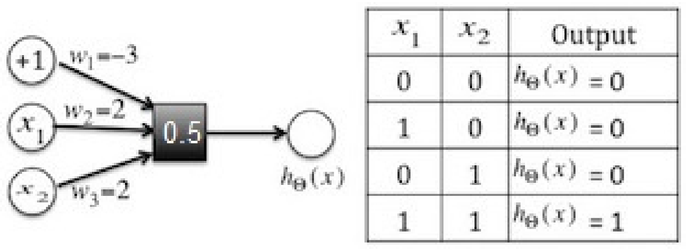
where x_0 is the bias unit. We have the following equations:

$$a_i^{(j)} = \sum_{k=1}^{m} w_{ik} x_k \tag{2.2}$$

Then $h_w(\vec{x}) = g \sum_{i=1}^{m} a_i^{(j)}$ finds the output result.

$$h_\theta(x) = \begin{pmatrix} a_1^2 \\ a_2^2 \\ \vdots \\ a_m^2 \end{pmatrix} = g(\theta) \begin{pmatrix} w_{10}^1 & w_{11}^1 & \cdots & w_{1m}^1 \\ w_{20}^1 & w_{21}^1 & \cdots & w_{2m}^1 \\ & \vdots & & \\ w_{m0}^1 & w_{m1}^1 & \cdots & w_{mm}^1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{pmatrix}$$

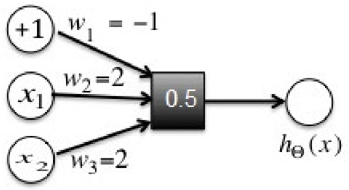
2.3.1 Logical Function ”AND”



x_1	x_2	Output
0	0	$h_\Theta(x) = 0$
1	0	$h_\Theta(x) = 0$
0	1	$h_\Theta(x) = 0$
1	1	$h_\Theta(x) = 1$

Figure 2.3: Conjunction

For the simple network with binary values $x_1 \in \{0, 1\}$ and $x_2 \in \{0, 1\}$, in order to compute the ”AND” logical function, we choose the appropriate weights which are shown in Figure 2.3. We compute the output function $h_\theta(x) = g(\theta)(w_1 + w_2 x_1 + w_3 x_2)$, where $g(\theta)$ is the step function. Using the binary values of x_1 and x_2, we calculate $h_\theta(x)$ and complete the truth table. The column on the right means $h_\theta(x)$ is approximating x_1 and x_2. The output is equal to one if $x_1 = 1$ and $x_2 = 1$, so $h_\theta(x) \approx x_1 \wedge x_2$.



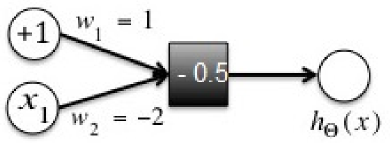
x_1	x_2	Output
0	0	$h_\Theta(x) = 0$
1	0	$h_\Theta(x) = 1$
0	1	$h_\Theta(x) = 1$
1	1	$h_\Theta(x) = 1$

Figure 2.4: Disjunction

2.3.2 Logical Function ”OR”

In Figure 2.4, by changing the connected weights we compute the disjunction, ”OR” logical function. So, $h_\theta(x) \approx x_1 \vee x_2$.

2.3.3 Logical Function ”NOT”



x_1	Output
0	$h_\Theta(x) = 1$
1	$h_\Theta(x) = 0$

Figure 2.5: Negation

In Figure 2.5, the output represent the negation, ”NOT”. To compute all complicated logical representation in multiple layered networks, we need at least one head of each layer. More generally, as we have neural networks with multiple hidden layers, we can imagine building up more and more complex functions by adding additional layers to the network. However, the challenge with multiple layers is that the learning problem becomes non–convex and we need both an approximate algorithm as well as sufficient computational power in order to train the parameters of the deep network.

2.4 Feedforward and Feedbackward Models

There are two different network structures, *feedforward* and *feedbackward*. A feedforward neural network is a type of network architecture where the connections are ”fed forward”, i.e., there is no cyclic connections and the output of any layer does not affect that same layer. This network structure has fixed inputs and outputs where traveling through the network starts from the input layer, then traverses through the hidden layers, and then ends at the output layer. They are broadly used in pattern recognition. Furthermore, they are a good candidate for any functional mapping problems where we want to know how a number of input variables affect output variable. This architecture is also referred to as *bottom–up* or *top–down*

There is, however, the feedbackward which is also called *backpropagation*. This has the training algorithm with three steps: (1) feed forward the values; (2) compute the error, and; (3) propagate it back to the earlier layers. More precisely, feedforward is part of the feedbackward algorithm and comes before backpropagation. Feedbackward networks can get highly complicated because signals in the networks can travel in both directions by introducing loops. Feedbackward networks are dynamic, i.e., their state is changing continuously until they reach an equilibrium point. Also, they stay at equilibrium until the input changes and a new equilibrium point is found.

Preliminaries

In this chapter we briefly recall some of the standard notations concerning logic programs with the main focus on propositional logic programs form [16, 12]. The more specialized definitions and notions are defined further in the related sections. For further background on notations regarding logic programs, cf. [12].

An *atom* is a propositional variable. A *literal* is an atom or a negated atom. In propositional logic program a *Horn clause* is defined: $q \leftarrow p_1, \ldots, p_n$, where q is an atom and is called the *head*, p_i are literals, and $\{p_1, \ldots, p_n\}$ is called the *body of the clause*. The clause body could be empty and there is a conjunctive relation between the literals of the clause body. The *clause length* is equal to the number of literals in the body of the clause. A clause is called *definite* if all literals p_i are atoms. A *logic program* $\mathcal{P}$ is defined as a finite set of rules and facts. A *normal propositional logic program* $\mathcal{P}$ over a propositional language $\mathcal{L}$ is a finite set of clauses; moreover, if all the clauses in the program $\mathcal{P}$ are definite, it is called a *definite propositional logic program*.

Let $\mathcal{B}$ be the set of all propositional variables occurring in $\mathcal{L}$. An interpretation I is a mapping $\mathcal{B} \to \{true, false\}$. We will usually identify I with the set of atoms (predicates) which it maps to *true*. $2^{\mathcal{B}}$ is the set of all interpretations. $\mathcal{I}_{\mathcal{P}}$ is the set of all interpretations of program $\mathcal{P}$. Note that $\mathcal{I}_{\mathcal{P}}$ is the *powerset* of the (finite) set $B_{\mathcal{P}}$ of all atoms occurring in $\mathcal{P}$ [16].

Definition 2. ***Reduced Logic Program:*** *A logic program $\mathcal{P}$ is called reduced if it holds the following:*

1. *For every clause $q \leftarrow p_1, ..., p_n$ in $\mathcal{P}$, all p_i are mutually distinct.*
2. *For any two rules $q \leftarrow p_1, ..., p_n$ and $q \leftarrow r_1, ..., r_m$ in $\mathcal{P}$, $\{p_1, ..., p_n\} \nsubseteq \{r_1, ..., r_m\}$, where $n \leq m$.*

Definition 3. ***Model:*** *A model of a clause C is an interpretation I which maps C to true (in symbols: $I \vDash C$). A model of a program $\mathcal{P}$ is an interpretation which maps every clause in $\mathcal{P}$ to* true *[18].*

Definition 4. ***Immediate Consequence Operator:*** *Let $\mathcal{B}$ be the set of all propositional variables occurring in propositional logic program $\mathcal{P}$. The operator $\mathcal{T}_{\mathcal{P}} : \mathcal{I}_{\mathcal{P}} \rightarrow \mathcal{I}_{\mathcal{P}}$ is called immediate consequence operator associated with program $\mathcal{P}$ such that*

$$\mathcal{T}_{\mathcal{P}}(I) := \{q \mid \textit{there is a clause } \{q \leftarrow \mathcal{B}\} \in \mathcal{P} \textit{ such that } I \models \mathcal{B}\}.$$

If $\mathcal{T}_{\mathcal{P}}(I) = I$, then I is a supported model and vice versa. This operator is well-known to be monotonic in the sense that whenever $I \subseteq J$, then $\mathcal{T}_P(I) \subseteq \mathcal{T}_P(J)$.

Rule Extraction

Even though the representation of symbolic knowledge is necessary for different applications of connectionist learning system, neural symbolic integration attempts to achieve connectionist processing of complex logical knowledge, learning, and inferences. Regarding the fact of the various applications of neural networks in real-world problems (e.g. in engineering, bioinformatics and robotics), knowledge extraction from the trained neural networks has been the core of the interests [3, 16]. The tendency to make the learned knowledge accessible to human interpretation and formal assessment necessitate the accuracy, efficiency, and rules comprehensibility of the methods for extracting symbolic knowledge from trained neural networks [3]. For each problem we need to extract knowledge in symbolic form. Then the symbolic extracted knowledge can be used for further processing using inference engines or other knowledge based systems. Intuitively, for a given trained neural network, the extraction task is to find the relations between inputs and outputs such that certain inputs cause a particular output. Extracting knowledge from trained networks mainly focus on rule extraction in propositional case.

In propositional rule extraction, we are looking for the sets of propositional rules (i.e., propositional Horn Clauses) which capture or approximate the input–output mapping $f : \mathcal{I} \rightarrow \mathcal{I}$, where $\mathcal{I}$ denotes the power set of the finite set $\mathcal{B}$ of all propositional variables corresponding to the nodes. In order to obtain such sets, there exists two main lines of approaches which mainly focus on first–order–neural symbolic integration. We use propositional logic program for this sake. The first one is a layer–by–layer approach and the second one is a neural networks approach as black boxes.

4.1 Propositional Rule Extraction Using the Layer-By-Layer Method

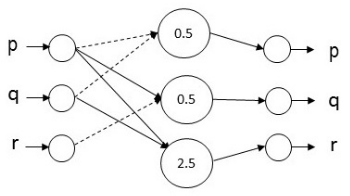
Among the learning strategies from neural network, Garcez [8] presented the connectionist inductive learning and logic programming ($C - IL^2P$) system, which is a massively parallel computational model. This model is based on a feedforward ANNs that integrates inductive learning from examples and background knowledge, with deductive learning from logic programming. This method starts with propositional logic program representation of background knowledge. It then applies a new translation algorithm which could generate a trained neural network with examples, as well as compute the stable model of the program inserted in it. Furthermore, extracting a revised logic program from the network uses the result of refining the background knowledge with training examples.

There is a trade–off between the complexity and quality of rule extraction methods. Rule extraction method of symbolic knowledge from trained neural networks introduced by Garcez and Broda [1] captured non-monotonic rules encoded in the network. This method is introspective and seeks to construct the rules out of the weights associated with the connections between nodes in the networks, usually proceeding in a layer-by-layer fashion under differentiable activation function [8, 1]. In their method they used the partial ordering on the input space along with a number of pruning and simplification rules that interact with such an ordering to ameliorate some of the problems of the rule extraction. Pruning rules reduce the search space of the input vectors so that we avoid clearing irrelevant input vectors, and the simplification rules decrease the size of the extracted set of rules. The method uses the pedagogical approaches to guarantee the knowledge extracted is equivalent to the network and the extraction process is sound and complete. However, the pedagogical approaches are not effective to use when the network size increases. In this case decomposition method resolve this limitation. We decompose the network to subsets of the weight of each neurons in the hidden and output layer of the network such that the neurons' input potential exceeds its threshold. This method decreases the complexity of the extraction algorithm to a large extend.

4.2 Propositional Rule Extraction as a Black Box

This method is in regard to a black box network and considers only the input–output function f. One of the main issues of this method was that the set of rules extracted from the networks were particularly redundant and more complex to interpret than the trained networks. Lehmann [16] showed how to obtain a unique irredundant representation of the extracted data from ANNs by using the propositional logic rule derivation. He proved that excluding the use of negation within the knowledge based is required in order to obtain the desired unique representation, i.e., by considering definite logic programs. However, in presence of the negation the unique representation is not expected. In both cases, definite logical program and normal logical program, extracting the irredundant representation from the trained network would be plausible by using some of the proper algorithms. Lehmann 's method was mainly based on learning cycles for the rules extraction: first, using background knowledge to initialize an untrained network; second, training the networks based on the background knowledge, and; lastly, extracting the data from the trained network. The black box approach ended with on interesting result. For every monotonic function $f : \{0,1\}^n \rightarrow \{0,1\}^m$, there exists a unique reduced set, as defined in definition 2, of positive propositional rules which capture exactly the function f. Reduced means no redundancies, and as small as possible.

4.2.1 Definite Logic Program



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

Figure 4.1: A Simple Propositional Logic Program and the Corresponding Network

In order to extract a logic program from the definite logic program, we activate the input layer of the given network with all possible interpretations to obtain the corresponding interpretation of the output layer. Interpretation by applying the immediate consequence operator leads to a monotone mapping $f : \mathcal{I}_{\mathcal{P}} \to \mathcal{I}_{\mathcal{P}}$ for the knowledge extracted from the given network. Referring to [16], for every such mapping, $f : \mathcal{I}_{\mathcal{P}} \to \mathcal{I}_{\mathcal{P}}$, by applying the *full exploration-definite* algorithm we can construct a propositional logic program $\mathcal{P}$ such that $\mathcal{T}_{\mathcal{P}} = f$. To construct a definite logic program $\mathcal{P}$ from this algorithm, for a given monotone mapping f, for each interpretation $I = \{p_1, p_2, \ldots, p_n\} \in I_{\mathcal{P}}$, we add the clause $r \leftarrow p_1, p_2, \ldots, p_n$ to $\mathcal{P}$ for each $r \in f(I)$. Furthermore, by employing a linear order $\prec$ using in *reduction algorithm*, we obtain program $\mathcal{Q}$ for $f = \mathcal{T}_{\mathcal{P}}$ such that $\mathcal{T}_{\mathcal{P}} = \mathcal{T}_{\mathcal{Q}}$. Program $\mathcal{Q}$ is also a reduced definite propositional logic program which is least in terms of size. The most interesting result about the reduction algorithm is $\mathcal{Q}$ extracted from this algorithm is an unique reduced definite propositional logic program that satisfies the mentioned properties.

Algorithm 1: Reduced Definite Program Extraction

```
Input: A monotone mapping $f : \mathcal{I}_1 \to \mathcal{I}_2$.
Output: $\mathcal{P}$, a definite logic program with $\mathcal{T}_{\mathcal{P}}(I) = f(I)$ for all $I \in \mathcal{I}_1$.
1: Initialization: $\mathcal{P} = \emptyset$.
2: Choose a total linear order $\prec$ on $\mathcal{I}_1$, such that for any $I_i, I_j \in \mathcal{I}_1$ with $i < j$
   we have $|I_i| < |I_j|$.
3: for all $I = \{p_1, \dots, p_n\} \in \mathcal{I}_1$, chosen in ascending order according to $\prec$ do
4:    for all $q \in f(I)$ do
5:       if there is no $q \leftarrow q_1 \wedge \cdots \wedge q_n$ in $\mathcal{P}$ with $\{q_1, \dots, q_n\} \subseteq I$ then
6:          add the rule $q \leftarrow p_1 \wedge \cdots \wedge p_n$ to $\mathcal{P}$.
7:       end if
8:    end for
9: end for
10: Return $\mathcal{P}$ as result.
```

The following theorem was shown in [16].

Theorem 1. *Let $f : \mathcal{I}_1 \to \mathcal{I}_2$ be monotonic. Then there exists a unique reduced logic program $\mathcal{P}$ with $\mathcal{T}_{\mathcal{P}} = f$. Furthermore, this logic program can be obtained using Algorithm 1.*

If we drop the precondition on f to be monotonic, then Theorem 1 no longer holds, because of the fact mentioned above that immediate consequence operators are always monotonic. The following corollary extracted from Theorem 1 shows the program obtained by Algorithm 1 is unique.

Theorem 2. *Given a logic program $\mathcal{P}$, there is always a unique reduced logic program $\mathcal{Q}$ with $\mathcal{T}_{\mathcal{P}} = \mathcal{T}_{\mathcal{Q}}$.*

Proof. Given $\mathcal{P}$, we know that $\mathcal{T}_{\mathcal{P}}$ is monotonic. Now apply Theorem 1. □

Theorems 1 and 2 show that the results extracted from definite logic program from a neural networks must be the same as the results of Algorithm 1. The following is an example of applying Algorithm 1 for the given definite logic program $\mathcal{P}$.

Example 1. *Let $\mathcal{B}_1 = \{p_1, p_2, p_3\}$ and $\mathcal{B}_2 = \{q_1, q_2\}$ be input and output sets, respectively, and consider the logic program $\mathcal{P}$ given as*

$$q_1 \leftarrow p_1 \wedge p_2$$

$$q_1 \leftarrow p_1 \wedge p_2 \wedge p_3$$

$$q_1 \leftarrow p_1 \wedge p_3$$

$$q_2 \leftarrow p_1$$

$$q_2 \leftarrow p_1 \wedge p_2.$$

Applying Algorithm 1 then yields the reduced program

$$q_1 \leftarrow p_1 \wedge p_2$$

$$q_1 \leftarrow p_1 \wedge p_3$$

$$q_2 \leftarrow p_1.$$

Here we mention some of the main principles of this model. There is one unit in the input and output layer of the network for each atom in the program, and there is a unit in the hidden layer for each clause. The connection between the layers are constructed such that the input-output behavior of the network matches the $\mathcal{T}_\mathcal{P}$ operator. Figure 4.1 shows a propositional logic program $\mathcal{P}$ and its network obtained by translation algorithm applied to $\mathcal{P}$. Connections with weight 1 are depicted *solid* and the ones with weight -1 are *dashed*. The numbers denote the threshold of the units. The knowledge represented by program $\mathcal{P}$ can essentially be captured by the operator $\mathcal{T}_\mathcal{P}$. The program contains the predicate letters p, q, r and we consider the interpretations related to these variables. For each rule in $\mathcal{P}$, a logical threshold unit is added to the hidden layer such that the unit become active if and only if the preconditions of the rule are met by current pattern of the input layer.

4.2.2 Normal Logic Program

As mentioned earlier it is possible to extract a normal logic program $\mathcal{P}$ from the given network such that the behavior of the associated $\mathcal{T}_\mathcal{P} - operator$ and the input-output mapping of the network are identical, but not necessarily a least one. At first we introduce some of the required definitions, then explain the method proposed by Lehmann [16] to extract all the reduced normal logic programs from the given networks.

Definition 5. ***Valid Clause:*** *Let $\mathcal{T}_\mathcal{P}$ be an immediate consequence operator, and r a predicate. $\mathcal{B} = p_1, p_2, \dots, \neg q_1, \dots, \neg q_b$ is called valid with respect to r and $\mathcal{T}_\mathcal{P}$ if and only if for every interpretation $I \subseteq \mathcal{B}_\mathcal{P}$ with $I \models \mathcal{B}$, we have $r \in \mathcal{T}_\mathcal{P}(I)$.*

Definition 6. ***Allowed Clause:*** *Let $\mathcal{T}_\mathcal{P}$ be an immediate consequence operator, and r a predicate. $\mathcal{B} = p_1, p_2, \dots, \neg q_1, \dots, \neg q_b$ is called allowed with respect to r and $\mathcal{T}_\mathcal{P}$ if the following properties hold:*

1. *$\mathcal{B}$ is valid with respect to $\mathcal{T}_\mathcal{P}$ and r.*

2. *There is no valid body $\mathcal{B}' \subset \mathcal{B}$ for r and $\mathcal{T}_\mathcal{P}$.*

In order to obtain the minimal normal programs for the given network, we could apply the *intelligent program search* algorithm. This algorithm is based on the full program search instead of using of heuristics, i.e., the score function, to add clauses to subprograms (cf. [16]). The intelligent program search algorithm can be applied to any arbitrary monotone mapping $f : I_\mathcal{P} \to I_\mathcal{P}$ over the set of predicates $\mathcal{B}_\mathcal{P} = \{q_1, q_2, \dots, q_m\}$. The goal is extracting a logic program $\mathcal{Q}$ such that in each iteration for each predicate $q_i \in \mathcal{B}_\mathcal{P}$, we construct the set S_i of allowed clause bodies for q_i. Then we add the set $\mathcal{Q}_i$ to $\mathcal{Q}$ where $\mathcal{Q}_i$ consists of the smallest clause body C in S_i such that the $score(C, \mathcal{Q})$ is maximum. The extracted normal program $\mathcal{Q}$ from the intelligent search algorithm cannot guarantee the uniqueness of the solution, but the output is the set of all allowed clause bodies with respect to $\mathcal{T}_\mathcal{P}$ and $\mathcal{B}_\mathcal{P}$.

We discussed algorithms for extracting definite and normal propositional logic programs from neural networks. The results show the existence and uniqueness of the solution for definite propositional logic program; however, for the normal case, it lost the uniqueness property and we could only expect extracting the least reduced normal programs from the networks. In Chapter 6 we only worked on definite logic program with adding background knowledge. We will see that the uniqueness property of the solution is no longer valid. Looking for the normal logic program is considered a future work.

Recurrent Neural Networks and Core Method

In this section we discuss knowledge based ANNs, i.e., networks which are initiating by available background knowledge before training methods are applied. The semantics of logic programs is defined as the least fixed point of an appropriate meaning operator. If program $\mathcal{P}$ is definite, this operator is called the *immediate consequence operator* $\mathcal{T}_{\mathcal{P}}$, which is monotonic function 3. Howeve, the $\mathcal{T}_{\mathcal{P}}$ operator for the normal program $\mathcal{P}$ could be non-monotonic and there is no guarantee of the existence of a least fixed point of $\mathcal{T}_{\mathcal{P}}$. Hölldobler in [13] used the metric method to establish a strong relationship between logic programming and connectionist models of computations. His research mainly focused on investigating the relationship between the normal propositional logic program and recurrent connectionist system.

5.1 Three–Layer Network

Holldobler-Kalinke at 1994 extended the approach by McCullch–Pitts and worked on the representation of propositional logic programs, and their semantics by using massively parallel reasoning. For any logical program $\mathcal{P}$ we construct a three–layer recurrent network that is updated along the implication, where $\mathcal{T}_{\mathcal{P}}$ represents the meaning (semantics) of program $\mathcal{P}$ through its fixed point. The recurrent network updates along layers corresponding to the iterator of the semantic operator. The fixed point of the operator (semantics of the program) can be computed in a parallel manner. Garcez–Zaverucha in 1999 [8] and Garcez–Broda-Gabby in 2001 [1] used the results of the core method for development of a learning paradigm. They established a new neural–symbolic learning cycle using a differentiable activation function and using the backpropagation algorithm. The neural–symbolic learning cycle is applied to the untrained neural network with initializing the initial background knowledge. We learn the neural network through a repeated cycle by modifying the initialization. We could extract the learned knowledge in the form of propositional rules after a finite number of training. The network function maps binary vectors, i.e., any input value 1 is interpreted as *true* and 0 as *false* and then outputs interpreted a value as true or false according to a threshold.

5.2 Recurrent Network and Core Method

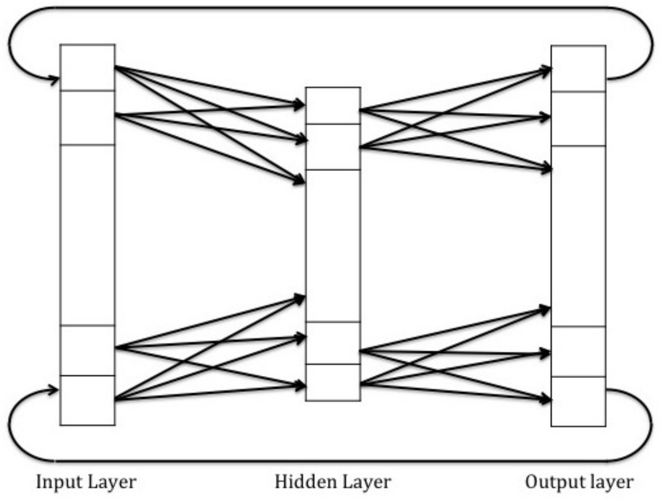


Figure 5.1: Thee–Layer Recurrent Connectionist Network

To construct recurrent networks we consider the so–called feedforward networks, which compute functions from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n and m are the number of input and outputs units, respectively. If the output units of a n–layer feedforward network $\mathcal{N}$ connects to the input units of $\mathcal{N}$, the constructed network is called a *recurrent network* [13, 4].

Let $\mathcal{L}$ be a logic language, where $\mathcal{T}_\mathcal{P}$ is the corresponding immediate consequence operator for the given logic program $\mathcal{P}$ under language $\mathcal{L}$. If $\mathcal{I}$ is the set of interpretations for $\mathcal{P}$, there is a mapping $\mathcal{R} : \mathcal{I} \to \mathbb{R}^n$. *Core* is a feedforward network computing $f_\mathcal{P} : \mathbb{R}^n \to \mathbb{R}^n$ such that the following holds:

$$\text{if } \mathcal{T}_\mathcal{P}(I) = J, \text{ then } f_\mathcal{P}(\mathcal{R}(I)) = \mathcal{R}, \text{where } I, J \in \mathcal{I}. \tag{5.1}$$

$$\text{if } f_\mathcal{P}(\vec{s}) = \vec{t}, \text{ then } \mathcal{T}_\mathcal{P}(\mathcal{R}^{-1}(\vec{s})) = \mathcal{R}^{-1}(\vec{t}), \text{ where } \vec{s}, \vec{t} \in \mathbb{R}^n. \tag{5.2}$$

Connect the units in the output layer recursively to the units in the input layer. $I = f_p(\mathcal{T}_\mathcal{P})$ if and only if the recurrent network converges to or approximates $\mathcal{R}(I)$. Connectionist model generation uses recurrent networks with feedforward core. Figure 5.1 shows a blueprint of a three–layer recurrent connectionist network. The left layer is a vector of input units and the right layer is a vector of output units, where the input and the output units are identical. Each unit in the input layer is connected to each unit in the output layer, and each unit in the output layer is connected to the corresponding unit in the input layer. The weights between all of the connectionist layers are equal. The subnet consisting of the three layers and the connections between the input and the hidden layers, as well as between the hidden and the output layers, is a three–layer feedforward network, which will be used to approximate the operator $\mathcal{T}_\mathcal{P}$ obtained from a certain program $\mathcal{P}$. In the recurrent network, the $\mathcal{T}_\mathcal{P}$ is computed with an arbitrary initial activation of the input layer with an initial interpretation. Multilayer feedforward networks are one of the most generally used and greatly investigated connectionist models.

Hölldobler [13] showed that *two–layer feedforward networks* cannot compute $\mathcal{T}_\mathcal{P}$ for definite $\mathcal{P}$. The multilayer feedforward neural network, i.e., at least one hidden layer, is the simplest network that can be used to compute the $\mathcal{T}_\mathcal{P}$ associated with definite propositional program $\mathcal{P}$. Therefore, for each program, $\mathcal{P}$ there is a multilayer feedforward network which computes $\mathcal{T}_\mathcal{P}$ and vice–versa.

The main idea of the *core method* is to use a feedforward connectionist network to compute the meaning function of logic programs. The propositional core method allows for model generation with respect to variety of logics in a connectionist system.Hölldobler [13] proved that for any propositional logic program $\mathcal{P}$, there exists a three–layer recurrent network, i.e, core, computing its meaning function $\mathcal{T}_\mathcal{P}$ and, for each acyclic logic program $\mathcal{P}$, there exists a core with the recurrent connectionist such that the computation with any arbitrary initial input converges and yields to a unique fixed point of $\mathcal{T}_\mathcal{P}$. Furthermore, if $\mathcal{P}$ is the program associated with three–layer feedforward subnet of the three–layer recurrent network $\mathcal{N}$, and $\mathcal{N}$ always settles down in a unique stable state, then $\mathcal{T}_\mathcal{P}$ has a unique fixed point. All of these results are extendable to three–valued or four–valued logics. In order to do this, we should extend the input and output layers of the network such that for each propositional variable $\mathcal{P}$ occurring in the formula, there are two units in the layers representing $\mathcal{P}$ and $\neg\mathcal{P}$, respectively. Program $\mathcal{P}$ obtained from the recurrent network is not always acceptable, and finding the stable state is not always possible. There is a close relationship between normal propositional logic programs and recurrent networks of binary threshold. We always could use the three–layer feedforward networks in order to find the approximation of $\mathcal{T}_\mathcal{P}$ for propositional program $\mathcal{P}$.

Rule Extraction with Background Knowledge

During the training process from the raw data, ANNs acquire expert knowledge about the problem, and the ability to recognize this knowledge similar to but previously unencountered situations in a way which often surpasses the abilities of human expert. One of the limitations in the range of the applicability of the neural networks technology is that the knowledge obtained during the training process is hidden within the acquired network architecture and connections weights, which is not directly accessible for analysis, reuse or improvement. As discussed in Section 4, rulesets extraction from either method are subject to be large and complex. From inspection of these rulesets, it is often difficult to obtain real insights into what the networks have learned. In this section, we investigate rule extraction under the assumption that there is additional background knowledge which can be connected to the network node activation, with the expectation that such background knowledge will make it possible to formulate simpler rulesets that still explain the input–output functions of the networks if the background knowledge is also taken into account. In the following, we consider a very simple example to illustrate a key insight as to how additional background knowledge can change the result proved by Lehmann [16] for the propositional logic program.

Example 2. *Assume there is an input–output mapping of program $\mathcal{P}$ of the neural networks without background knowledge. Then*

$$\mathcal{P} = \{o \leftarrow p_1 \wedge p, o \leftarrow p_2 \wedge p\}$$

and there is background knowledge program K with rules

$$\mathcal{K} = \{q \leftarrow p_1, q \leftarrow p_2\}.$$

Then, taking background knowledge into account, we obtain the simplified input–output mapping $\mathcal{P}_{\mathcal{K}}$, as

$$\mathcal{P}_{\mathcal{K}} = \{o \leftarrow q \wedge p\}.$$

In Example 2 q serves as a *more general* propositional, and p_1 and p_2 are the sub-branches of q, e.g., p_1 could stand for ”is an apple”, and p_2 could stand for ”is a banana”, while q could stand for ”is a fruit”. Now we could take, e.g., p to stand for ”is ripe”, and o to stand for ”can be harvested”, then we obtain a not–so–abstract toy example, where the background knowledge facilitates a simplification because it captures both apples and bananas using the more general concept ”fruit”.

As discussed in Section 4.2.1, Lehmann [16] proved the uniqueness of the reduced definite propositional logic program $\mathcal{Q}$ for any $\mathcal{T}_{\mathcal{P}}$ of a definite propositional logic program $\mathcal{P}$. In this section, we investigate and discuss in detail to what extent we can carry over results regarding positive rulesets from [16] to new scenarios with background knowledge.

We now introduce more notation as needed and recall preliminary results from [16].

6.1 Definitions and Notation

We recall some of the results from [16] and previously used notation in Section 3 which will be central for this section.

Following the notation presented in Section 3, let $\mathcal{B}$ be a finite set of propositional variables, let $\mathcal{I}$ be the power set of $\mathcal{B}$, and consider functions $f : \mathcal{I} \to \mathcal{I}$ as discretizations of input-output functions of trained neural networks. In this section, we only examine all of the (propositional) logic programs with positive (or definite) rules.

We need to make a mild assumption. We assume that the propositional variables used to represent input and outputs node are distinct, i.e., each propositional variable represents either an input node or an output node, but not both. This means that the power set $\mathcal{B}$ can be partitioned into two sets, $\mathcal{B}_1$ and $\mathcal{B}_2$, such that $\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_2$. As a consequence, we obtain $\mathcal{I}_1$ and $\mathcal{I}_2$, the corresponding power sets, such that the semantic operator (immediate consequence operator) is a function $\mathcal{T}_\mathcal{P} : \mathcal{I}_1 \to \mathcal{I}_2$.

Due to our assumption, we now give a different but equivalent formalization for the definition of the immediate consequence operator, which will help us in this section. For any $I = \{p_1, \ldots, p_n\} \subseteq \mathcal{B}$, let $c(I) = p_1 \wedge \cdots \wedge p_n$. Now, for a given logic program $\mathcal{P}$ and $I \in \mathcal{I}_1$, we obtain:

$$\mathcal{T}_\mathcal{P}(I) = \{q \in \mathcal{B}_2 \mid c(I) \wedge \mathcal{P} \models q\},$$

where $c(I) \wedge \mathcal{P}$ denotes $c(I) \bigwedge_{R \in \mathcal{P}} R$.

Definition 7. ***Equivalence Class:*** *Two programs, $\mathcal{P}$ and $\mathcal{G}$, are in* equivalence class *if $\mathcal{T}_\mathcal{P}(I) = \mathcal{T}_\mathcal{G}(I)$, where I is an interpretation, and symbolically we denote it by $\mathcal{P} \sim \mathcal{G}$.*

6.2 Extracting Reduced Definite Program with Background Knowledge

As laid out in Theorems 1 and 2 of Section 4.2.1, for any definite logic program $\mathcal{P}$, there is a unique reduced logic program. In this section, we investigate Theorem 1 when considering additional background knowledge. Assume $\mathcal{P}$ is a logic program which captures the input–output function of a trained network according to Theorem 1. Let logic program $\mathcal{K}$ account for our background knowledge, which may use additional propositional variables, i.e., propositional variables not necessarily occurring in $\mathcal{P}$. We then seek a logic program $\mathcal{P}_{\mathcal{K}}$ such that, for all $I \in \mathcal{I}_1$, we have

$$\{q \in \mathcal{B}_2 \mid c(I) \wedge \mathcal{P} \models q\} = \{q \in \mathcal{B}_2 \mid c(I) \wedge \mathcal{K} \wedge \mathcal{P}_{\mathcal{K}} \models q\}.$$

In this case, we call $\mathcal{P}_{\mathcal{K}}$ a *solution* for $(\mathcal{P}, \mathcal{K})$.

6.2.1 Existence of Solutions

There are two assumptions for the rules in program $\mathcal{K}$. The first assumption is no propositional variable from $\mathcal{B}_2$ appears in $\mathcal{K}$. The second assumption propositional variables from $\mathcal{B}_1$ appear only in the body of rules in $\mathcal{K}$. The justification for the first assumption is based on the explanation of the networks such that the occurrence of the variables from $\mathcal{B}_2$ in $\mathcal{K}$ would bypass the network. The second one can be easily justified by the use case, which indicates that the network input activation should be our starting point, i.e., the activation should not be altered by the background knowledge. The existence of the solution cannot be guaranteed if we drop the second assumption, i.e., appearance of the propositional variables from $\mathcal{B}_1$ in the body of rules in $\mathcal{K}$.

Example 3. *Let $\mathcal{B}_1 = \{p_1, p_2\}$, let $\mathcal{B}_2 = \{q_1, q_2\}$. Then, for the given programs $\mathcal{P}$*

$$q_1 \leftarrow p_1$$

$$q_2 \leftarrow p_2$$

and $\mathcal{K}$

$$p_2 \leftarrow p_1,$$

there is no solution for $(\mathcal{P}, \mathcal{K})$. To see this, assume $\mathcal{P}_{\mathcal{K}}$ is a solution for $(\mathcal{P}, \mathcal{K})$. Then, because $p_2 \wedge \mathcal{P} \models q_2$, we obtain $p_2 \wedge \mathcal{K} \wedge \mathcal{P}_{\mathcal{K}} \models q_2$. But then $p_1 \wedge \mathcal{K} \wedge \mathcal{P}_{\mathcal{K}} \models q_2$ although, $p_1 \wedge P \not\models q_2$. In other words, $\mathcal{P}_{\mathcal{K}}$ cannot be a solution for $(\mathcal{P}, \mathcal{K})$.

However, the existence of the solution has been guaranteed under the second assumption.

Proposition 1. *Let $\mathcal{P}$ and $\mathcal{K}$ be the given logic programs under our standing assumptions. Then there always exists a solution for $(\mathcal{P}, \mathcal{K})$ which is reduced.*

Proof. Because rule heads from $\mathcal{K}$ never appear in $\mathcal{P}$, we obtain

$$\{q \in \mathcal{B}_2 \mid c(I) \wedge \mathcal{P} \models q\} = \{q \in \mathcal{B}_2 \mid c(I) \wedge \mathcal{K} \wedge \mathcal{P} \models q\}$$

for all $I \in \mathcal{I}_1$, i.e., $\mathcal{P}$ is always a solution for $(\mathcal{P}, \mathcal{K})$. Existence of a reduced solution then follows from Theorem 2. $\square$

Proposition 1 shows the existence of the solution. However, this solution is not always unique. Naturally, we are interested in all *other* solutions which are simpler that $\mathcal{P}$.

Proposition 2. *There exists logic programs $\mathcal{P}$ and $\mathcal{K}$ which satisfy our standing assumptions, such that there are two distinct reduced solutions for $(\mathcal{P}, \mathcal{K})$.*

Proof. Let $\mathcal{B}_1 = \{p_1, p_2, p_3\}$ and $\mathcal{B}_2 = \{q\}$. Then consider the programs $\mathcal{P}$

$$q \leftarrow p_2 \wedge p_3$$

$$q \leftarrow p_1 \wedge p_3$$

and $\mathcal{K}$

$$r_1 \leftarrow p_2 \wedge p_3$$

$$r_1 \leftarrow p_1 \wedge p_3$$

$$r_2 \leftarrow p_1$$

$$r_2 \leftarrow p_2.$$

The two logic programs

$$\mathcal{P}_{\mathcal{K}_1} = \{q \leftarrow r_1\}$$

and

$$\mathcal{P}_{\mathcal{K}_2} = \{q \leftarrow p_3 \wedge r_2\}$$

are then both reduced solutions for $(\mathcal{P}, \mathcal{K})$. □

For any given logic program $\mathcal{P}$ and program $\mathcal{K}$ of additional background rules, the number of solutions extracted by applying Algorithm 1 is actually exponential in the worst

Algorithm 2: Construct All Reduced Solutions for (P, K)

```
Input: Logic programs $\mathcal{P}$ and $\mathcal{K}$ with $\mathcal{T}_{\mathcal{P}} : \mathcal{I}_1 \to \mathcal{I}_2$ and $T_K : \mathcal{I}_1 \to \mathcal{I}_3$ which satisfy
       our standing assumptions, where $\mathcal{B}_2 = \{q_1, \ldots, q_n\}$.
Output: All the reduced solutions for $(\mathcal{P}, \mathcal{K})$.
 1: Set $\mathcal{S} = \emptyset$ and $\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_3$.
 2: Set $\mathcal{I}$ to be the power set of $\mathcal{B}$.
 3: Set $\mathcal{R}$ to be the power set of $\mathcal{I}$.
 4: for all $(R_1, \ldots, R_n) \in \mathcal{R}^n$ do
 5:   for all $i \in \{1, \ldots, n\}$ do
 6:     $Q_i = \{c(B) \to q_i \mid B \in R_i\}$
 7:   end for
 8:   Set $Q = \bigcup_{i \in \{1,\ldots,n\}} Q_i$.
 9:   if $T_Q$ is a solution for $(\mathcal{P}, \mathcal{K})$ then
10:     Apply Algorithm 1 to $T_Q$ to obtain a reduced program $S$ with $T_S = T_Q$.
11:     if $S \notin \mathcal{S}$ then
12:       Add $S$ to $\mathcal{S}$.
13:     end if
14:   end if
15: end for
16: Return $\mathcal{S}$ as result.
```

case. We will discuss about it in the proof of Theorem 3.

6.2.2 Algorithms

Let $\mathcal{P}$ and $\mathcal{K}$ be the given propositional logic programs under our standing assumptions in Section 6.2.1. In order to compute all reduced solutions for $(\mathcal{P}, \mathcal{K})$, i.e., $\mathcal{P}_{\mathcal{K}}$, we use Algorithm 2, which uses a brute-force approach to check all possible logic programs that can be constructed over the given propositional variables, where they constitute all the solutions for $(\mathcal{P}, \mathcal{K})$. To obtain a corresponding reduced program for every extracted solution from Algorithm 2, it invokes Algorithm 1. The output of the Algorithm 2 is a set of all reduced solutions for $(\mathcal{P}, \mathcal{K})$.

Algorithm 2 has exponential runtime because in line 4, if the set $\mathcal{B}$ has m elements, the power set $\mathcal{I}$ has 2^m interpretations and therefore the power set $\mathcal{R}$ has 2^{2^m} interpretations. If there are n propositional variables in $\mathcal{B}_2$, then the searching cost to find all of the solutions for $(\mathcal{P}, \mathcal{K})$ is $O(n.2^{2^m})$. Even though Algorithm 2 is quite correct and always terminates, it is too naive to be practically useful for anything other than toy examples.

Theorem 3. *The problem of finding all solutions to $(\mathcal{P}, \mathcal{K})$ is, in the worst-case, exponential, in the combined size of $\mathcal{P}$ and $\mathcal{K}$.*

Proof. Let n be any positive integer. Define the logic program $\mathcal{P}_n$ to consist of the single rule

$$q \leftarrow p_1 \wedge \cdots \wedge p_n$$

and let

$$\mathcal{K}_n = \{r_{i,1} \leftarrow p_i, r_{i,2} \leftarrow p_i \mid i = 1, \ldots, n\}.$$

Then, for any function $f : \{1, \ldots, n\} \rightarrow \{1, 2\}$, the logic program

$$\mathcal{P}_f = \{q \leftarrow r_{1,f(1)} \wedge \cdots \wedge r_{n,f(n)}\}$$

is a reduced solution for $(\mathcal{P}_n, \mathcal{K}_n)$. Since there exist 2^n distinct such functions f, the number of reduced solutions in this case is 2^n, so their production is exponential in n, while the combined size of $\mathcal{P}_n$ and $\mathcal{K}_n$ grows only linearly in n. $\square$

Algorithm 2 computes all of the reduced solutions to $(\mathcal{P}, \mathcal{K})$. However, Algorithm 3, which is the combination of Algorithms 1 and 2 can find only one of the reduced solutions which makes it more efficient.

Algorithm 3: Reduced Solution for $(\mathcal{P}, \mathcal{K})$

```
Input: Logic programs $\mathcal{P}$ and $\mathcal{K}$ with $\mathcal{T}_{\mathcal{P}} : \mathcal{I}_1 \to \mathcal{I}_2$ and $\mathcal{T}_{\mathcal{K}} : \mathcal{I}_1 \to \mathcal{I}_3$ which satisfy
       our standing assumptions.
Output: A reduced solution for $(\mathcal{P}, \mathcal{K})$.
 1: Set $S = \emptyset$ and $\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_3$.
 2: Set $\mathcal{I}$ to be the power set of $\mathcal{B}$.
 3: Choose a total linear order $\prec$ on $\mathcal{I}$, such that for any $I_i, I_j \in \mathcal{I}$ with $i < j$ we
    have $|I_i| < |I_j|$.
 4: for all $I = \{p_1, \dots, p_m\} \in \mathcal{I}_1$, chosen in ascending order according to $\prec$ do
 5:   for all $q \in \mathcal{T}_{\mathcal{P}}(I)$ do
 6:     if $c(I) \wedge \mathcal{K} \wedge S \not\models q$ then
 7:       Set endloop = false.
 8:       Choose first $J = \{b_1, \dots, b_n\} \in \mathcal{I}$ according to $\prec$.
 9:       while endloop = false do
10:         if $c(I) \wedge \mathcal{K} \wedge S \wedge (J \to q) \models q$ then
11:           if $\{H \in \mathcal{I}_1 \mid H \wedge \mathcal{K} \wedge S \wedge (q \leftarrow J) \models q\} \subseteq \{H \in \mathcal{I}_1 \mid q \in \mathcal{T}_{\mathcal{P}}(H)\}$ then
12:             Add the rule $q \leftarrow J$ to $S$ and set endloop = true.
13:           end if
14:         else
15:           Choose next $J = \{b_1, \dots, b_n\} \in \mathcal{I}$ according to $\prec$.
16:         end if
17:       end while
18:     end if
19:   end for
20: end for
21: Return $S$ as a result.
```

Proposition 3. *Algorithm 3 is correct and always terminating.*

Proof. Like Algorithm 1, Algorithm 3 checks all combinations of $I \in \mathcal{I}_1$ and $q \in \mathcal{T}_{\mathcal{P}}(I)$ and makes sure that there are rules in the output program such that $c(I) \wedge \mathcal{K} \wedge S \models q$. The rules for the output program are checked one by one in increasing length until a suitable one is found. Note that the rule $q \leftarrow I$ is going to be checked at some stage, i.e., the algorithm will either choose this rule, or a shorter one, but in any case we will eventually have $c(I) \wedge \mathcal{K} \wedge S \models q$. This shows that the algorithm always terminates and that we obtain $c(I) \wedge \mathcal{K} \wedge S \models q$ for all $q \in \mathcal{T}_{\mathcal{P}}(I)$.

In order to demonstrate that the algorithm output S is indeed a solution for $(\mathcal{P}, \mathcal{K})$, we also need to show that for all $q \in \mathcal{B}_2$ and $H \in \mathcal{I}_1$ we have $H \wedge \mathcal{K} \wedge S \models q$ implies $q \in \mathcal{T}_{\mathcal{P}}(H)$. This is in fact guaranteed by line 11 of Algorithm 3, i.e. the algorithm output S is indeed a solution for $(\mathcal{P}, \mathcal{K})$.

We finally show that the output of the algorithm is reduced. Assume otherwise. Then there are $q \leftarrow I_1$ and $q \leftarrow J$ in S with $I_1 \subsetneq J$. By our condition on the order, we thus have $I_1 \prec J$, and so we know that $q \leftarrow I_1$ was added to S earlier in the algorithm than $q \leftarrow J$. Now let us look at the instance of line 12 in Algorithm 3 when the rule $q \leftarrow J$ was added to S. In this case, (using notation from the algorithm description, and S denoting the current S at that moment) we know that $c(I) \wedge \mathcal{K} \wedge S \wedge (J \rightarrow q) \models q$ and $c(I) \wedge \mathcal{K} \wedge S \not\models q$. This implies $c(I) \wedge \mathcal{K} \wedge S \models J$, and because $I_1 \subseteq J$ we obtain $c(I) \wedge K \wedge S \models I_1$. But we also observed that $I_1 \rightarrow q$ is already contained in S at this stage, and thus we obtain $c(I) \wedge \mathcal{K} \wedge S \models q$, which contradicts the earlier statement that $c(I) \wedge \mathcal{K} \wedge S \not\models q$. Therefore, we have to reject the assumption that S is not reduced; hence S is indeed reduced. This completes the proof. $\square$

The following example indicates how Algorithm 2 computes all of the reduced solutions of $(\mathcal{P}, \mathcal{K})$.

Example 4. *Let $\mathcal{B}_1 = \{p_1, p_2, p_3\}$ and $\mathcal{B} = \{q_1, q_2, q_3, q_4\}$. Consider the program $\mathcal{P}$*

$$\begin{array}{ll} q_1 \leftarrow p_1 & q_1 \leftarrow p_2 \\ q_2 \leftarrow p_1 & q_2 \leftarrow p_2 \\ q_3 \leftarrow p_1 & q_3 \leftarrow p_2 \\ q_4 \leftarrow p_1 & q_4 \leftarrow p_2 \wedge p_3 \end{array}$$

and $\mathcal{K}$

$$\begin{array}{ll} r_1 \leftarrow p_1 & r_1 \leftarrow p_2 \wedge p_3 \\ r_2 \leftarrow p_1 & r_2 \leftarrow p_2 \\ r_3 \leftarrow p_2. & \end{array}$$

Then there is only one reduced solution $\mathcal{P}_\mathcal{K}$ for $(\mathcal{P}, \mathcal{K})$, which is

$$\begin{array}{ll} q_1 \leftarrow r_2 & q_2 \leftarrow r_2 \\ q_3 \leftarrow r_2 & q_4 \leftarrow r_1. \end{array}$$

Note, that $\mathcal{P}_\mathcal{K}$ is simpler and shorter than $\mathcal{P}$.

Referring to Proposition 3, we expect to have more than one reduced solution for $(\mathcal{P}, \mathcal{K})$. Even though we miss the uniqueness of the result by adding background knowledge, all the solutions are in equivalent class.

Proposition 4. *Let $\mathcal{P}$ and $\mathcal{K}$ be the given programs under out standing assumption. If there exist more than one reduced solution for $(\mathcal{P}, \mathcal{K})$, all the solutions are in equivalent class.*

Proof. Let there be n distinct reduced solutions for $(\mathcal{P}, \mathcal{K})$. We want to show there is a class of solutions $\mathcal{P}_{\mathcal{K}}$ such that $[\mathcal{P}_{\mathcal{K}}]_R = \{\mathcal{P}_{\mathcal{K}_m} \in S | \mathcal{P}_{\mathcal{K}_i} \sim \mathcal{P}_{\mathcal{K}_m}, 1 \leq i \leq n\}$, where S is the set of all reduced solutions, such that $c(I) \wedge \mathcal{P} \vDash q$ if and only if $c(I) \wedge [\mathcal{P}_{\mathcal{K}}]_R \wedge \mathcal{K} \vDash q$. We use contradiction to prove it.

Let there be two classes $[\mathcal{P}_{\mathcal{K}}]_R$ and $[\mathcal{P'}_{\mathcal{K}}]_R$ such that if $c(I) \wedge \mathcal{P} \vDash q$, then $c(I) \wedge [\mathcal{P}_{\mathcal{K}}]_R \wedge R \vDash q$ and $c(I) \wedge [\mathcal{P'}_{\mathcal{K}}]_R \wedge \mathcal{K} \vDash q$.

Without loss of generality, assume the rules of the two classes are all the same except with one different rule, $[\mathcal{P}_{\mathcal{K}}]_R = \{q \leftarrow \mathfrak{r}_1, \mathfrak{r}_2, ...\mathfrak{r}_{k-1}, \mathfrak{r}_k, \mathfrak{r}_{k+1}, ..., \mathfrak{r}_l\}$ and $[\mathcal{P'}_{\mathcal{K}}]_R = \{q \leftarrow \mathfrak{r}_1, \mathfrak{r}_2, ...\mathfrak{r}_{k-1}, \rho, \mathfrak{r}_{k+1} ..., \mathfrak{r}_l\}$, $l \geq 1$. For the two classes the only different variables are $\mathfrak{r}_k$ and ρ. As $[\mathcal{P}_{\mathcal{K}}]_R$ is equal to one of the members in set S, suppose $[\mathcal{P}_{\mathcal{K}}]_R = \{\mathcal{P}_{\mathcal{K}_i}\}$:

Case 1. Suppose $\rho \in \mathcal{P}_{\mathcal{K}_i}$. Then $\mathfrak{r}_k = \rho$, and there is nothing to prove.

Case 2. Suppose $\rho \notin \mathcal{P}_{\mathcal{K}_i}$. Then $\mathfrak{r}_k \neq \rho$ and $\exists \mathcal{P}_{\mathcal{K}_j} \in S$ such that $\rho \in \mathcal{P}_{\mathcal{K}_j}$. As all the programs $\mathcal{P}_{\mathcal{K}_i} \in S$, $1 \leq i \leq l$, it does not effect the uniqueness property.

Case 3. Suppose ρ is a rule that is not in any member programs in set S. Then ρ is one of the redundant clauses which should be dropped from the class of $[\mathcal{P'}_{\mathcal{K}}]_R$. $\square$

Example 5. *Let $\mathcal{B}_1 = \{p_1, p_2, p_3\}$ and $\mathcal{B}_2 = \{q\}$. Consider programs $\mathcal{P}$*

$$q \leftarrow p_2$$

$$q \leftarrow p_1 \wedge p_3$$

and $\mathcal{K}$

$$r_1 \leftarrow p_2 \wedge p_3$$

$$r_1 \leftarrow p_1 \wedge p_3$$

$$r_2 \leftarrow p_2$$

Then then are two reduced programs, $\mathcal{P}_{\mathcal{K}_1}$ and $\mathcal{P}_{\mathcal{K}_2}$, for $(\mathcal{P}, \mathcal{K})$, which are:

$$\mathcal{P}_{\mathcal{K}_1} = \{q \leftarrow p_2; q \leftarrow r_1\}$$

and

$$\mathcal{P}_{\mathcal{K}_2} = \{q \leftarrow r_1, q \leftarrow r_2\}$$

The set $S = \{\mathcal{P}_{\mathcal{K}_1}, \mathcal{P}_{\mathcal{K}_2}\}$, and

$$[\mathcal{P}_{\mathcal{K}}]_R = \{q \leftarrow r_1, q \leftarrow r_2\}$$

is the unique class for the given programs.

Conclusions and Further Work

We reviewed the propositional rule extraction from trained neural networks for definite and normal programs. Then we investigated the propositional rule extraction from trained neural networks under background knowledge for the case of definite rules. We showed that the extraction of the reduced logic program with additional background knowledge is guaranteed by applying some of the mild assumptions to the background knowledge and also by using a monotonic input–output function. This extracted solution exactly reproduced the input-output function. In addition, we showed the reduced extracted solution is not unique anymore; however, all the reduced extracted solutions are in equivalence class. Furthermore, we presented algorithms for obtaining corresponding reduced programs.

The investigated results can be fundamentals for further work, rather than directly applicable in practice. Our observation that background knowledge can yield simpler extracted rulesets of course carries over to more expressive logics which extend propositional logic.

It is such extensions which we intend to pursue, which hold significant promise for practical applicability: structured information on the World Wide Web, as discussed in the Introduction, is provided in logical forms which are usually non-propositional fragments of first-order predicate logic, or closely related formalisms. In particular, description logics [2], i.e. decidable fragments of first-order predicate logic, form the foundation of the Web Ontology Language OWL. First-order rules are also commonly used [14]. This raises the question how to extract meaningful non-propositional rules from trained neural networks while taking (non-propositional) background knowledge, in a form commonly used on the World Wide Web, into account.

Bibliography

[1] D.M. Gabbay A.S. dAvila Garcez a, K. Broda. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125:155–207, 2001.

[2] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2nd edition, 2010.

[3] Sebastian Bader and Pascal Hitzler. Dimensions of neural-symbolic integration – A structured survey. In Sergei N. Artëmov, Howard Barringer, Artur S. d'Avila Garcez, Luís C. Lamb, and John Woods, editors, *We Will Show Them! Essays in Honour of Dov Gabbay, Volume One*, pages 167–194. College Publications, 2005.

[4] Sebastian Bader and Steffen Hlldobler. The core method: Connectionist model generation. In *Lecture Notes in Computer Science*, volume 4132, pages 1–13, 2006.

[5] D. Beckett, T. Berners-Lee, E Prud'hommeaux, and G. Carothers. *RDF 1.1. Turtle – Terse RDF Triple Language*. W3C Recommendation, 25 February 2014. Available at http://www.w3.org/TR/turtle/.

[6] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[7] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data – The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[8] Artur S. d'Avila Garcez and Gerson Zaverucha. The connectionist inductive lerarning and logic programming system. *Applied Intelligence*, 11(1):59–77, 1999.

[9] Ramanathan V. Guha, Dan Brickley, and Steve Macbeth. Schema.org: evolution of structured data on the web. *Commun. ACM*, 59(2):44–51, 2016.

[10] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph, editors. *OWL 2 Web Ontology Language Primer (Second Edition)*. W3C Recommendation, 11 December 2012. http://www.w3.org/TR/owl2-primer/.

[11] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. CRC Press/Chapman & Hall, 2010.

[12] Pascal Hitzler and Anthony K. Seda. *Mathematical Aspects of Logic Programming Semantics*. CRC Press/Chapman and Hall, 2010.

[13] Steffen Hölldobler, Yvonne Kalinke, Fg Wissensverarbeitung Ki, Fakultat Informatik, and Tu Dresden. Towards a new massively parallel computational model for logic programming. In *In ECAI94 workshop on Combining Symbolic and Connectioninst Processing*, pages 68–77, 1991.

[14] Adila Krisnadhi, Frederick Maier, and Pascal Hitzler. OWL and rules. In Axel Polleres, Claudia d'Amato, Marcelo Arenas, Siegfried Handschuh, Paula Kroner, Sascha Ossowski, and Peter F. Patel-Schneider, editors, *Reasoning Web. Semantic Technologies for the Web of Data – 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures*, volume 6848 of *Lecture Notes in Computer Science*, pages 382–415. Springer, 2011.

[15] Maryam Labaf, Pascal Hitzler, and Anthony B. Evans. Propositional rule extraction from neural networks under background knowledge. *International Workshop on Neural-Symbolic Learning and Reasoning, NESY'17*, pages 1–10, July 2017.

[16] Jens Lehmann, Sebastian Bader, and Pascal Hitzler. Extracting reduced logic programs from artificial neural networks. *Appl. Intell.*, 32(3):249–266, 2010.

[17] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[18] Uwe Schoning. *Logic for Computer Science*. Springer, 1989.

[19] Geoffrey G. Towell and Jude W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine learning*, 13(3):71–101, 1993.

[20] Denny Vrandecic and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[21] McCulloch W.S. and Pitts W. Bltn. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 52(1/2):99–115, 1990.

[22] Jing T. Yao. Knowledge extracted from trained neural networks: What's next? *Proc. SPIE 5812, Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security*, 2005 151 (April 05 2005); doi:10.1117/12.604463.

[23] ZhiHua Zhou, Yuan Jiang, and ShiFu CHen. Extracting symbolic rules from trained neural network ensembles. *AI Communications*, 16(1):3–15, 2003.