UNIVERSITÄT OSNABRÜCK

INSTITUTE OF COGNITIVE SCIENCE

Bachelor's Thesis

An Approach to Supervised Learning of Three-Valued Łukasiewicz Logic in Hölldobler's Core Method

Frederik Harder

July 27, 2015

First supervisor:	Prof. Dr. Kai-Uwe Kühnberger
Second supervisor:	Tarek Richard Besold, PhD

Acknowledgements

I would like to thank Tarek R. Besold who supervised this thesis and provided valuable guidance throughout all stages of the project.

Additional thanks go to Sebastian Höffner for this LaTeX template.

Contents

Chapter 1

Introduction

The field of neural-symbolic integration attempts to bridge the gap between two prominent paradigms in artificial intelligence. Symbolic AI, the first of the two, encompasses explicit knowledge representation, logic programming and search-based problem solving techniques which have been responsible for many of the early successes in artificial intelligence such as game playing, automated theorem proving and natural language processing [14, 21, 24]. While the paradigm is still very much alive in expert systems managing and reasoning over vast quantities of symbolic data, it is also at times referred to as good old-fashioned AI, having lost some of its appeal as its limitations have become apparent. Learning from, and finding structure in sets of noisy data is something symbolic AI largely fails at. Unfortunately this means that whole classes of problems which are integral to a common conception of intelligence, such as image and voice recognition, can not be addressed using symbolic AI.

The second paradigm is that of machine learning. As the name suggests, it refers to a variety of methods for learning from data, artificial neural networks (ANN) being one prominent group of these methods. Aided by a leap in processing power and available data, machine learning has been credited with most of the more recent accomplishments in AI, from the now commonplace feat of handwriting recognition to self-driving cars and the fully autonomous learning of computer games [20, 4, 19]. Promising as the paradigm may be, there are areas in which, on its own, it performs very poorly. While the learning of simple logical dependencies from data is achieved with relative ease, the process becomes increasingly difficult when higher order concepts are involved [10]. Because knowledge is represented in a distributed fashion that is hard to interpret from an outside perspective, it can also be difficult to provide background knowledge in a format which the machine learning algorithm can use. Both are problems that often become trivial when tackled with a symbolic system.

Much stands to be gained from a unification of the two paradigms that could cancel out their respective weak spots and highlight their strengths. Neural-symbolic integration offers some ideas in how this may be achieved, centering around the concept of the neural-symbolic cycle.

The cycle contains two reasoning systems. One is symbol-based, utilizing available expert knowledge, and the other is a connectionist system or ANN, which learns from data. The challenge of interfacing these systems is twofold. Coming from the symbolic side, the first task

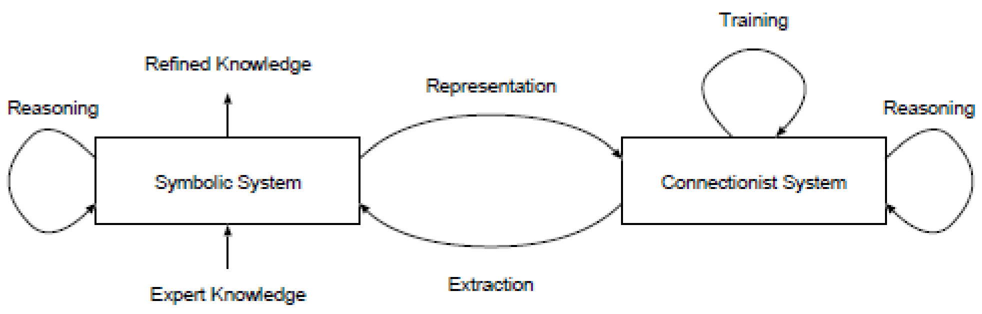

Figure 1.1: The neural-symbolic cycle (from [2])

is to find a way of translating the existing symbolic knowledge into the connectionist system, finding a representation that is appropriate for the network. Secondly, one needs to devise methods for extracting the information gained by the connectionist system through learning and convert it back into a clean symbolic format. Equipped with these processes of representation and extraction the system as a whole is capable of incorporating both background knowledge and training data as either become available.

It should be made clear, that the task of constructing such a cycle rapidly increases in difficulty with raising the expressive capacities of the involved systems. There are approaches for fragments of first order logic [11, 3], but most results focus on various propositional logics. Furthermore, extraction algorithms for connectionist systems tend to be intractable. So while the general method of the field can be described in a few pages, the underlying problems are hard and there is still a long way to go before neural-symbolic integration may be applied to state-of-the-art methods of either paradigm.

The *core method*, introduced by Steffen Hölldobler in [12], has since been developed as a neural-symbolic system for, among others, propositional modal [9] and covered first order logic programs [3]. It provides a way of translating logic programs into a type of multilayer perceptron (MLP) which, embedded in the core architecture, computes least models of these programs. In their 2009 paper Logics and Networks for Human Reasoning [13], Steffen Hölldobler and Carroline Dewi Puspa Kencana Ramli discuss a variant of the core method for three valued Łukasiewicz logic and its applicability to cognitive modelling tasks. In the discussion of their work, the authors make the claim that the architecture they have used can be modified in such a way, as to allow training via the backpropagation algorithm [22]. Further, the application of rule extraction methods should allow closure of the neural symbolic cycle. The aim of this thesis is to put these ideas into practice by providing a modified core suitable for supervised learning, implementing and executing supervised learning with the backpropagation algorithm and, finally, constructing a rule extraction method. Chapter 2 of this thesis will give an overview of the theory and methods underlying this work, after which chapter 3 is used for an in-depth documentation of the project. Results are shown in chapter 4, followed by a discussion in chapter 5. Algorithms and lengthy proofs have been relegated to the appendix.

Chapter 2

Foundations

As basis for this work, a number of methods and terminology should be clarified. The three sections of this chapter will give a short introduction to Łukasiewicz logic programs, Hölldobler's core method as well as the backpropagation algorithm. Some basic familiarity with classical logic and neural networks is assumed.

2.1 Łukasiewicz logic programs

Three-valued Łukasiewicz logic

Łukasiewicz Logic was proposed in its ternary version in 1917 by Polish philosopher and logician Jan Łukasiewicz [16], as a result of his work on modalities in logic. The addition of a third value was meant to introduce a notion of possibility and indeterminism to logical reasoning. Metaphysical import aside, this logic was the first one to break with the true/false dichotomy of classical logic and thus lay the basis for the development of further many-valued and fuzzy logics thereafter. The connective definitions are provided below, following the notation used by Hölldobler et al. A noteworthy property, when compared to Kleene logic, for example, is the definition of $u \to u$ and $u \leftrightarrow u$ as true. This allows for the existence of tautologies, i.e. formulas which are true under all interpretations, and preserves some of those familiar from classical logic. Conventionally, an interpretation assigns one of the three values $\top$, $\bot$ and u to each atom in the Universe U. In the given context it makes sense to define an interpretation as a tuple $I = \langle I^\top, I^\bot \rangle$, where $I^\top$ is a set containing all atoms assigned the value $\top$ and $I^\bot$ contains all atoms assigned $\bot$. No atom is in both sets and those assigned u are in neither set. One can speak of an *empty* interpretation when $I^\top \cup I^\bot = \varnothing$ and a *partial* interpretation when $I^\top \cup I^\bot \subsetneq U$. I is a *model* of a formula G, iff $I(G) = \top$.

$\wedge$	$\top$	u	$\bot$
$\top$	$\top$	u	$\bot$
u	u	u	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\vee$	$\top$	u	$\bot$
$\top$	$\top$	$\top$	$\top$
u	$\top$	u	u
$\bot$	$\top$	u	$\bot$

	$\neg$
$\top$	$\bot$
u	u
$\bot$	$\top$

$\rightarrow$	$\top$	u	$\bot$
$\top$	$\top$	u	$\bot$
u	$\top$	$\top$	u
$\bot$	$\top$	$\top$	$\top$

$\leftrightarrow$	$\top$	u	$\bot$
$\top$	$\top$	u	$\bot$
u	u	$\top$	u
$\bot$	$\bot$	u	$\top$

For Hölldobler, Łukasiewicz logic is of interest for modelling empirical observations of human reasoning. Specifically, he seeks to provide a logical model for the suppression task experiment by cognitive scientist Ruth Byrne [5]. Byrne has analysed, what conclusions readers will typically draw from a certain class of natural language statements. Here is one example. When reading the statement *"If Marian has an essay to write, she will study late in the library. She has an essay to write."*, 96% of all subjects concluded that *"Marian will study late in the library."* Later on, this item appeared again with the added information that *"If the library stays open, she will study late in the library."*, which led only 38% of participants to conclude that Marian will indeed study in the library. It can be observed, that additional information led many participants to revoke their previous inferences, even though this information was not contradictory. The non-monotonicity of this reasoning suggests that it cannot be modelled by classical logic and Hölldobler and Dietz argue [6], that three-valued Łukasiewicz logic interpreted under weak completion, which will be explained shortly, fits the findings best.

Logic programs

Logic programs are defined as a finite set of clauses of the form $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$ where the head of the clause, A, is an atom and the B_i, with $1 \leq i \leq n$, in the body are either literals, $\top$ or $\bot$. Clauses of the form $A \leftarrow \top$ and $A \leftarrow \bot$ are called *positive* and *negative facts* respectively.

These logic programs are interpreted under weak completion, which takes a logic program and transforms it into one single formula, thereby changing how it is evaluated. Firstly, the bodies of all clauses with the same head are concatenated as a disjunction into one body. After this step, the resulting formulas consist of one implication per head. Subsequently, all $\leftarrow$ are replaced with $\leftrightarrow$. As a result, atoms which are heads in clauses whose bodies all evaluate as $\bot$ are now $\bot$ as well. Finally, concatenating all clauses into one conjunction creates a single formula representing the weakly completed program.

Weak completion adds non-monotonicity to Łukasiewicz logic. Atoms which evaluate as $\bot$ because all associated bodies evaluated as $\bot$, can become $\top$ when adding another clause without contradiction. Also, weakly completed Łukasiewicz logic programs are never contradictory, always having at least one model [13].

Consequence operators

Models for such a logic program P can be computed through a consequence operator Φ_P. Starting from an empty interpretation I, the immediate consequence $\Phi_P(I)$ is calculated as a new interpretation and this process is iterated, until $I = \Phi_P(I)$ and a fixed point is reached. Hölldobler and Kencana Ramli have shown that the least models of Łukasiewicz logic under weak completion are identical to the least fixed points of a consequence operator $\Phi_{SvL,P}$ devised by Stenning and van Lambalgen [23], which is defined as follows:

$\Phi_{SvL,P}(I) = \langle J^\top, J^\perp \rangle$, where
$J^\top = \{A \mid \exists (A \leftarrow \mathit{body}) \in P : I(\mathit{body}) = \top\}$ and
$J^\perp = \{A \mid \exists (A \leftarrow \mathit{body}) \in P \wedge \forall (A \leftarrow \mathit{body}) \in P : I(\mathit{body}) = \perp\}$

Hölldobler et al. provide an algorithm which translates the $\Phi_{SvL,P}$ consequence operator of a given program into a 3-layer feed-forward network, which computes the same function. Like the consequence operator, this network may then be used on multiple iterations until a fixed point is reached.

2.2 Hölldobler's core method

In the following, a detailed description of how the core architecture is set up will be provided. This account chooses a somewhat different perspective than that of the translation algorithm provided in [13]. While the algorithmic description is optimal for implementation, the angle used here will hopefully provide a better understanding of the core structure with regard to the modifications that must be made to it, and to the introduction of sigmoidal activation units in particular.

The network as a collection of logic gates

In both input and output layer of the network, each atom A of the program is represented by two neurons. Activation in the first one indicates $A = \top$, while activation in the second means $A = \perp$. If neither neuron is active, then $A = u$. The core does not allow for both neurons to be active in the same iteration. The input layer also contains one neuron each, representing $\top$ and $\perp$, which are always active. Each program clause, or rather each clause body, is represented by two neurons $\langle h^\top, h^\perp \rangle$ in the hidden layer. Whether a clauses body is mapped to $\top$, $\perp$ or u is encoded in the same way as was used for the atoms.

All connections between the layers of the core serve the function of logic gates. An $h^\top$ neuron is connected to one input layer neuron for each conjunct in the clause body it represents. If the conjunct is an atom A, it connects to $A^\top$, if it is a negated atom $\neg A$, it connects to $A^\perp$ and if the conjunct is $\top$, it connects to that unit. Connection weights and activation threshold are set to form an 'and'-gate, requiring activation of all input layer neurons for the clause neuron to fire. In case a conjunct is $\perp$, no connection is formed, but for sake of the logic gate this is treated as a connection to an inactive unit, preventing the clause neuron from ever firing. Respectively, an $h^\top$ neuron connects to $A^\perp$ neurons, where A is a conjunct and $A^\top$ neurons, where $\neg A$ is one. If $\perp$ is a conjunct, $h^\top$ connects to the $\perp$ neuron and if $\top$ is a conjunct, no connection is formed. Weights and threshold are set to form an 'or'-gate, such that $h^\top$ is activated when one or more input neurons fire. This way, clause bodies are represented as $\top$ if and only if all their conjuncts are mapped to $\top$ and represented as $\perp$, if and only if one or more conjuncts are $\perp$.

In the output layer, each neuron has one connection for each clause in which the associated atom appears as head. $A^\top$ neurons are connected to the $h^\top$ neurons of the associated clauses, forming an 'or'-gate and $A^\perp$ neurons are connected to the $h^\perp$ neurons, forming an 'and'-gate.

Thus atoms are $\top$ when one or more associated clauses are $\top$, and $\bot$, when all associated clauses are $\bot$.

The logic gates are implemented such that all connection weights in the network have the same value $\omega > 0$ and 'or'-gate thresholds are at $0.5 \cdot \omega$, while 'and'-gate thresholds equal to $(l - 0.5) \cdot \omega$, where l is the number of incoming connections. All neurons use the Heaviside activation function, emitting 1, if the received activation meets or exceeds the threshold and 0 otherwise. Given this setup, computing a fixed point merely involves feeding the network's output back into the input layer until it equals the previous output[1].

2.3 The backpropagation algorithm

The backpropagation algorithm, introduced by Rumelhart, Hinton and Williams in 1986 [22], has become the probably most widely used training algorithm for feed-forward networks. It offers a computationally efficient way of deriving the partial derivatives of the cost function for classification error with respect to each weight of the network. The partial derivatives are then used for adjusting the weights, usually through gradient descent, so the cost function is minimized.

The name backpropagation derives from the order in which the partial derivatives are calculated. This process begins in the output layer and propagates backward, layer by layer, as the calculation in each layer requires the results of its successor.

Variations of the algorithm

The derivation of the backpropagation algorithm is fairly general and holds for different cost and activation functions. Typical cost functions are the quadratic cost function[2] and the logistic cost function[3], the latter of which is used here. As activation function, the standard sigmoid is used. The specific formulas used in the algorithm depend on the choice of function. A derivation for the algorithm with quadratic loss function, along with a general introduction to the algorithm, can be found in [18] and an analogous derivation for the backpropagation that was used here is provided in the appendix.

The implementation uses on-line training, which means that weights are updated every time after calculating the error for one randomly selected sample. The alternative to this is batch training, where weights are updated only after the accumulated error for all samples has been calculated. On-line learning has been used because it better accommodates the large variations in sample size that are encountered in different cores. Aside from this, the choice is of no conceptual importance.

[1] The number of iterations necessary for reaching the least fixed point can be shown to be lesser or equal to the number of atoms. The network has no inhibitory connections, so more input always generate equal or more output. Starting from an empty interpretation, each subsequent iteration must therefore activate at least one additional unit, one per atom at maximum, or stop.

[2] $J(\vec{w}) = \frac{1}{2m}\sum_{i=1}^{m}(h_{\vec{w}}(x^{(i)}) - y^{(i)})^2$, $\vec{w}$ being the vector of weights and $h_{\vec{w}}(x^{(i)})$ being the output produced by the network with sample input $x^{(i)}$ as compared to sample output $y^{(i)}$

[3] $J(\vec{w}) = \frac{1}{m}\sum_{i=1}^{m}[(y^{(i)}\log h_{\vec{w}}(x^{(i)}) + (1 - y^{(i)})\log(1 - h_{\vec{w}}(x^{(i)}))]$, following the same notation

Modifications and improvements

Depending on the classification problem at hand, the backpropagation algorithm can perform badly in a number of ways. It is prone to getting stuck in local minima of the cost function with much higher classification error than the global optimum. The speed of convergence depends on the learning rate parameter, the value of which can pose some problems. A small value will lead to slow convergence in flat regions of the parameter space. Large values can lead to overshoot of optima in steep regions. As usual, overfitting of the data can become a problem as well. Each of these issues can be addressed with a variety of modifications to the algorithm.

Momentum saves the weight adjustment terms in each iteration and adds a fraction of them to the weight adjustment in the next iteration. This tends to speed up convergence by preventing fluctuation of the weights to some extent and also leads to some robustness against small local optima. As it can be implement with little effort, it has also been used in the experimental code.

Gradient descent may be replaced with more sophisticated optimization methods such as conjugate gradient which also speed up convergence. It is also possible to choose the weight initializations more carefully, allow for multiple starting points or partially randomize optimization in order to find better optima. For this, methods such as linear-least-squares initialization and simulated annealing may be used [7, 1]. Overfitting can be counteracted by factoring connection weights into the cost function, encouraging sparser connectivity and thus simpler models. This method is known as regularization.

The methods touched on here should provide a small overview of what steps can be taken to improve the performance of backpropagation when putting more effort into it. It should also illustrate, why the project discussed here focuses on qualitative rather than quantitative results. Where a consistent, if limited, level of learning success can be shown with a basic implementation, it is plausible to assume that further attempts with more sophisticated versions of the learning algorithm will yield much better results.

Chapter 3

Theory and implementation of learning cores

This chapter documents the conceptual work that has gone into this project, beginning with modifications made to the core architecture, followed by some remarks on the learning algorithm and a thorough discussion of the proposed rule extraction algorithm. It closes with a list of control measures used in the implementation.

Before supervised learning can be attempted in cores, three problems have to be addressed. The core architecture must reach fixed points to compute results. While the existence of these fixed points has been proven for translated programs, this result does not generalize to cores whose weights have changed over the learning process. The first task, therefore, is to ensure the existence of fixed points throughout the learning process. In addition, a differentiable activation function must be introduced, while preserving the core's semantics. The backpropagation algorithm relies on the computation of derivatives of the cost function which includes the activation functions of the network. The Heaviside function is not differentiable and must be replaced. Lastly, one must decide, what kinds of samples will be used for supervised learning. The core, as it was proposed by Hölldobler is only capable of computing the least fixed point, when starting from an empty interpretation. If one wants to capture any of the structure of the program, more than one sample is needed for training.

3.1 Ensuring a fixed point with unipolar weights

Convergence to a fixed point is essential to the core method. While this property is guaranteed for Hölldobler's discrete cores and will be proven for their sigmoidal analogs below, it is difficult to ensure it throughout the learning period, where the network may change drastically and with little regard for the structure in which it is embedded.

Convergence, therefore, should be guaranteed by something other than the initial setting of the weights. A possible solution to this issue, and the one employed here, is to restrict the network to unipolar weights. When limiting all non-bias weights to positive values, there are no inhibitory connections and thus the activation of the network will monotonically increase on every iteration until it plateaus at a fixed point.

The elimination of inhibitory units does of course reduce the modelling capacity of the network. The reason it can be done in good conscience here, is that the translation of logic programs into cores itself only uses positive weights and thus ensures that every Łukasiewicz logic program to be learned by a core can be fully modelled, and therefore also learned, using these simpler unipolar networks.

A standard activation function used in feed forward networks is the sigmoid function $sig(z) = 1/(1+e^{-z})$ where $z = w \cdot x$, the dot product of the weight vector and the incoming activations. In the implementation unipolarity is achieved by squaring all but the bias weight in the activation function. So, while the sigmoid function remains the same, z is now computed as $w_0 \cdot x_0 + \sum_{i>0}(\frac{1}{2}(w_i)^2 \cdot x_i)$. The values stored in the weight matrix may still be negative, but will effectively be treated as positive. To preserve the previous behaviour of cores, all non-bias weights are replaced by their respective square root after the translation algorithm has been applied. With this measure, the translation algorithm can ignore the modification to the activation function and act as if it was the standard sigmoid, so long as it only sets positive weights. The subsequent argument that semantics are preserved in the sigmoidal core will also assume the standard activation function.

3.2 Preserving core semantics

With the introduction of sigmoidal activation to the network, the range of possible activations for each neuron changes from 0 and 1 to the interval $]0, 1[$ and what it means for a neuron to 'fire' becomes less clear. To ensure compatibility with the core architecture, the network's output is discretized by rounding it half up to 0 and 1. A fixed point is reached when this rounded output is equal to the input of the current iteration. Whithin the network, however, instead of an activation value it makes more sense to define an interval bounded by a certain value $[A^+, 1]$ where all activation values in that interval are considered as firing, and another interval $[0, A^-]$ of activations regarded as not firing.

As these two intervals should be disjoint, it follows that $A^- < A^+$ and because the classification into firing and non-firing is integral to the way the core is built, it must be ensured that no activations in the interval $]A^-, A^+[$ are produced. Given these changes, the approach of using logic gates, which was explained in the previous chapter, can be maintained, but must deal with the following complications. Because the output of non-firing neuron is no longer 0 and can take on values up to A^- an 'or'-gate must ensure that it won't fire, even if all connected neurons send an activation of A^- each, while at the same time guaranteeing that it will fire if one neuron sends activation A^+ and all others send nothing. Similarly 'and'-gates should fire when all connected neurons from the previous layer send A^+, but not if all but one send an activation of 1 and the last one sends A^-. It becomes clear that these constraints of maximal non-activating input and minimal activating input (the terms are used here in a different context than later on in rule extraction) can only be satisfied with the right choice of A^- and A^+.

Setting omega

In the core, both A^- and A^+ are determined by the value of ω. If ω is large, A^- and A^+, approach 0 and 1 respectively. For a small ω, both values lie close to 1/2. It can be shown, that semantics of the network are preserved if $\omega > 2\log(2deg - 1)$, where deg is the maximum in-degree among neurons in the output layer. The formal proof for this can be found in the appendix.

3.3 Fixed point calculation with initial activation

The core architecture as defined by Hölldobler et al. serves to compute fixed points for a given logic program and no additional input. Evidently, this one sample of (empty) input and corresponding output does not contain exhaustive information about the program which produced it. To train a network which captures the functionality of the program, more samples are required. Given the context of logic programs, it seems like an obvious choice to generate additional samples for possible interpretations of the atoms. There are 3^n possible interpretations for a set of ternary logic formulas P with n atoms. What additional inferences P allows, based on a partial interpretation, provides information about P, and having this information for all 3^n interpretations specifies P to its semantic equivalence class[1].

C-interpretation

A naïve approach for using such partial interpretations in a core is to enforce them as the base activation while running the core, and see what additional inferences are drawn before reaching a fixed point. This is achieved by adding the interpretation to every starting activation on the first iteration as well as to the output at the end of every iteration.

This method must be called naïve because this definition of interpretations, while applicable to Łukasiewicz logic, actually makes very little sense for the weakly completed logic programs at hand. Determining the value of an atom from the outset, while leaving the program as is, takes away both the non-monotonicity and the property of non-contradiction. On the plus side, only few changes to the core are necessary to accommodate this interpretation, which will from now on be referred to as C-interpretation.

Ł-interpretation

In order to preserve the semantics of weakly completed Łukasiewicz logic, an alternative Ł-interpretation is proposed. Here the process will be handled slightly differently, as it seems more adequate to model different interpretations in such a way that they represent logic programs in their own right. As such, setting an atom A to $\top$ or $\bot$ in the interpretation should have the same effect as adding a positive or negative fact to the program. Doing this preserves the important property, that the Stenning-van-Lambalgen consequence operator always reaches a model. Note that in this choice of interpretation setting atoms to false only has an effect, if they do not occur as heads of clauses in the program, and that setting atoms to true in the interpretation will prevent the consequence operator from inferring these atoms to be false even if all other clauses in the program would lead to this conclusion.

[1] If the interpretation leads to no contradictions, it is a model of P, otherwise it is not. Knowing all possible interpretations of P provides the full extension.

Going with the interpretation as adding clauses to the program, the most intuitive approach would be adding neurons to the hidden layer of the core which represent these rules. Unfortunately this does not seem to be a viable option. The addition of neurons would change the in-degrees of some of the core's output units which would in turn necessitate an update of their respective bias weights, in order to maintain Łukasiewicz semantics. For each interpretation there could be changes to the whole network which would not only be computationally costly but also pose problems in the context of learning, where changes to the core network should likely be limited to the learning algorithm itself.

Instead it seems more promising to adjust the way in which the inputs to the network are generated and outputs are interpreted. For negative facts like $A \leftarrow \bot$ this is fairly easy. Given weak completion these clauses only affect the program at all, if there is no other clause with head A in the program. In this case A will be set to $\bot$ and keep this value, as there is no other clause to change it. This can be modelled in the core by checking the in-degree of the atom's associated output unit in the network. If the in-degree is 0, A is set to $\bot$ in the input to the network on every iteration as well as on the final output, which in the neural net means the activation of the neurons corresponding to A is $(0, 1)$. Positive facts of the type $A \leftarrow \top$ have to be treated differently. If such a clause exists, A will be true independently of the rest of the program. This means A should be set to $\top$ on all inputs as well as the final output, i.e. the activation is set to $(1, 0)$. Because the clause is part of the interpretation and not translated into the network, the network will still produce activation in the $A^{\bot}$ output neuron, whenever all program clauses with A in the head are false. The arising contradiction must be resolved and the easiest way of doing this is to set activation of the $A^{\bot}$ neuron in the output to 0 on all inputs and the final output.

C*-interpretation

In addition, a form of Ł-interpretation will be tested, which is different only in that it leaves out the contradiction resolution. This C*-interpretation is of interest because, as will be shown in the results, it can be trained better than Ł-interpretation but still bears some similarities. Training under C- and C*-interpretations performs so similarly that C-interpretation will not be discussed separately in the empirical results.

It is clear that all three interpretations generate the same output for empty interpretations. Furthermore it can be shown, that all non-contradictory models under C*-interpretation are equivalent to the Ł-interpretation under the same input[2]. All three interpretations have been implemented and tested.

3.4 Backpropagation in cores

With the modifications to the core that have been described above it is now possible to create samples and test the core's capacity for learning. In the given set-up, two cores are used. The first core is generated by translating the complete program into it and is subsequently run with all possible inputs computing the desired outputs, the pairs of which will be used as training

[2]A sketch of this proof is provided in the appendix

samples. Core number two is created based on a partial version of the program, where some clauses have been deleted. The learning task now, is to train the second core with samples from the first one and see whether it can learn the missing parts of the program.

It may take the core multiple iterations to reach a fixed point, but only the last one is used for training. For a given sample, the core is run on the sample input and when reaching a fixed point returns activation values of all the networks layers. Note that the activation of the output layer is not the final output of the core, which may contain modifications from interpretation or contradiction-resolution. The backpropagation algorithm is then applied to the network with that activation.

Due to the non-classical activation function used in the network, the algorithm differs slightly from its more common form. The derivation of the relevant formulas found in the appendix is done analogous to the proof of standard backpropagation found in [18].

3.5 Rule extraction

The algorithm for extracting information from a core discussed in this section focuses on C- and C*-interpretation. It has been pointed out previously that through iterations the core's activation increases monotonically under these interpretations, due to a lack of inhibitory connections in the network. The same reasoning ensures monotonicity with regard to interpretations. While the number of possible interpretations rises exponentially with the number of atoms, diminishing hopes for a tractable rule extraction algorithm, the property of monotonicity allows for heavy pruning, making a viable solution at least for small sample sizes possible.

In constructing the algorithm, Garcez et al.'s paper on knowledge extraction and their algorithm for regular networks [8] have been a big help and a good starting point. Still, the method presented here warrants an independent introduction as well as analysis for soundness and completeness.

The basic extraction algorithm

The algorithm extracts all minimal activating and all maximal non-activating inputs for each output neuron of the network, which can then be used to compute the logical rules generating this activation. In the following, the set of all inputs to the network will be looked at as a partial order with the input vector of zeros as bottom element and the vector of ones as top element. Input vectors are ordered in such a way that $v_1 \geq v_2 \Leftrightarrow \forall i : v_1[i] \geq v_2[i]$.

For each output neuron separately, the algorithm traverses the space of all possible interpretations by advancing alternatingly an upper and a lower boundary of interpretations starting from top and bottom element. The new boundaries are generated by computing all *direct successors* of each element of the existing boundary. For an element of the lower boundary a direct successor is a copy of the element in which exactly one activation is changed from 0 to 1. The direct successor of an element in the upper boundary, analogously, has one active input less than that element. All inputs connected through a series of direct successions will be called successors and the definition for predecessors follows from this. An input in the lower boundary is said to be *subsumed* by an activating input if it is a successor of that input and is subsumed by a non-activating input, if it is a predecessor of that input. For subsumption in the upper boundary, successor and predecessor relations are reversed. In either case, the target-activation produced by the subsumed input is equal to, and therefore determined by, the other input. Whenever an activating input is found in the lower boundary, which is not subsumed by an input already stored in the set of minimal activating inputs, it is added to that set. The progression through a lower boundary ensures that all predecessors have already been checked and the one that has been found is in fact minimal. Also, if all direct successors of a non-activating input are activating, then that input is added to the list of maximal non-activating inputs. In the upper boundary, relevant inputs are found in an analogous manner, where activation is the default. The algorithm terminates once the two boundaries have passed by one another, which is not implemented explicitly, but a result of the pruning mechanisms discussed below.

Soundness and completeness without pruning

Prior to pruning, the soundness and completeness of the extraction algorithm is evident, but spelled out here for the sake of completeness. All activating inputs found in the lower boundary, which are not greater than previously found ones are minimal, as all smaller activating inputs would be in that set. And all non-activating inputs whose direct successors are activating are maximal, as all their successors are activating due to monotonicity. The analog holds for the upper bound. Thus the extraction is sound. All minimal activating inputs are found by the algorithm, as they are activating and not subsumed by any other activating inputs. All maximal non-activating inputs are found by the algorithm as the successors of each are activating by definition. Here, too, the analog is true for the upper bound and so the extraction is complete as well. Later it will be shown that both properties are preserved under pruning rules.

Pruning motivation

Due to the monotonic nature of the space of possible inputs, once an activating input is found in the lower bound, none of its successors need to be investigated any more, as all of them will be activating as well. An efficient extraction algorithm must therefore limit its exploration of the space of possible inputs to the relevant nodes which may hold new information. The pruning is best explained from the perspective of one of the boundaries. From the perspective of the lower boundary, minimal activating inputs will be called *rules* and maximal non-activating inputs are called *anti-rules* (for lack of a better term). These terms are relative to the boundary, such that rules in the lower boundary are anti-rules in the upper boundary and vice versa. When a rule is discovered, all of its successors should be pruned, as their values will hold no new information. This must be ensured both in the current boundary, where it is a rule, and the opposite boundary, where it is an anti-rule and two pruning mechanisms ensure this.

Pruning details

The pruning mechanisms can not be explained without covering the specifics of the algorithm in some detail. It may help to have a look at the code provided in the appendix for reference.

To avoid too much confusion, the algorithm refers to inputs as vertex objects, owing to the graph-like feel of the partial order. Alongside its input value and a number of other things, each vertex stores a memory array to keep track of the direct successors it should generate and those that should be pruned. This array has one entry for each neuron, which is 1, if switching the value of this input from 0 to 1 or vice versa will generate a valid successor, 0 if the successor and all subsequent successors are invalid, and -1 if the direct successor is invalid, but later ones may be valid.

The *test* function checks whether a vertex is a rule or subsumed by an anti-rule. If the vertex is a rule, the memory array is set to all zeros, so that no successors are generated and the vertex is added to the set of found rules. If the vertex is subsumed by an anti-rule, some, but not necessarily all successors will be subsumed as well. In all places where switching the input would generate a subsumed direct successor, the memory array is set from 1 to -1.

This information is utilized in the *successors* function. As the name suggests, this function creates the direct successors of a given vertex, but also uses the step to exchange pruning information among the successors. Firstly, the input vertex is *tested* again, in case new anti-rules were discovered while traversing the other boundary. Then, each valid direct successor of the vertex (as indicated by the memory array) is looked up and if it does not exist yet, is generated and tested. For each such successor which is a rule, the preceding vertex's memory is set to 0 at the index, which was used to generate that successor, indicating that this successor should not be investigated further. After this has been completed, all the successors are traversed for a second time and all 0s from the vertex memory are copied into their respective memories as well. This way, each successor receives information about all vertices, with which it shares a common direct predecessor.

Now, when a rule is discovered, all of its direct predecessors will set the index in their memory which generated this rule to 0 and pass this information on to all their successors. If a vertex subsumed by the rule were to be generated, it would have to have a direct predecessor which is not subsumed by the rule (or the problem propagates down until this condition occurs). This predecessor, however, must be a successor of one of the rule's predecessors. Therefore it would not generate that vertex and it follows that no vertices subsumed by rules are generated. Note that the same does not hold for anti-rules.

Finally, the successors function also serves to determine, whether the given vertex is an anti-rule. This is the case, if the vertex is not subsumed by an anti-rule (i.e. no entry in the memory is set to -1) and no generated successor has the same target-activation value as the vertex. As rules trivially share these properties by having all their successors pruned, they must be filtered out. This is done by checking for the right target-activation, given the vertex's boundary, before adding it to the set of rules. In the lower bound an anti-rule must be non-activating, and activating, if in the upper bound. So now, when the *rules_for_target* function finally creates the two boundaries and traverses the partial order, the pruning ensures no vertices that are subsumed by known rules are looked at.

The anti-rule related pruning is handled with some care by the algorithm, as it can lead to problematic cases. In general, it might happen, that all direct successors of a vertex are subsumed by some anti-rules. In such a case, declaring all direct successors invalid could hinder the generation of valid ones down the line. If, for example, $(1,1,1,1,0,0,0)$ and $(1,0,0,0,1,1,1)$ were known anti-rules, the lower boundary input $(1,0,0,0,0,0,0)$ would generate no direct successors and unsubsumed successors like $(1,0,0,1,1,0,0)$ would not be reached. In the algorithm this problem is solved by looking only at the first anti-rule found, rather than the whole set of subsuming anti-rules. Apart from the trivial case where the anti-rule is the top or bottom element (which ends the search as either all inputs are activating or none are), a single anti-rule, will not prune all successors of the vertex. The indices at which the anti-rule differs from its predecessors (of which, barring the trivial cases, it has at least one) can be changed in the vertex to generate valid successors.

Of course this strategy can, at times, dismiss useful information and generate vertices which are subsumed by known rules at the point of creation. As a stand-in for more elaborate methods it will suffice to generate some first results.

Soundness and completeness with pruning

In order to ensure that soundness and completeness are maintained, it must be proven that pruning neither changes the results of examining a particular vertex, nor prevents any rule vertices from being examined. Addressing part one, the test for rules functions in the same way as without pruning and only relies on the vertex's target activation value, which is not affected by pruning. With pruning, the test for anti-rules employs the memory of the given vertex, rather than looking at all successors, but it does so, only to infer the target-activation values of the invalid immediate successors. Assuming that rules have been identified correctly, every 0 in the memory is linked to a successor which has a different target-activation value than the vertex. Each -1 is linked to a successor which has the same value as the vertex. Generating each of these invalid successors would take more time but yield the same result. Therefore all examined vertices are still classified correctly. The second part follows from the soundness of the pruning algorithm.

3.6 Controls

Verifying the correctness of an implementation as a whole beyond the checking of test cases is usually associated with an enormous effort and has therefore not been in the scope of this project. Checks have been installed at two crucial steps in the program which are worth mentioning.

Discrete core comparison

A function has been implemented which can run a core with discrete activation units as in Hölldobler's original algorithm (with the one difference that it squares all non-bias weights to compensate for the fact that they were reduced to their square roots in the new translation). The function is used to run a core with both kinds of activation and for all possible inputs under a chosen interpretation, returning an error if the activations reach different models for the same input. This way, it is possible to verify that the implementation of the translation algorithm with sigmoidal units does preserve the semantics of the cores.

Gradient checking

As a standard measure to ensure the correctness of the implementation of the backpropagation algorithm, a method called gradient checking has been implemented. This method approximates the partial derivatives of the cost function with respect to all weights in the network and compares its results to those of backpropagation. For each weight w in the network, gradient checking computes the cost function of two manipulated versions of the network with the same input. In one version the value of w is increased by a small value ε, in the other it is decreased by that same ε. For small ε the line defined by these two values of the cost function $J(w)$ approximates the gradient of $J(w)$ at the original value of w. Thus $(J(w-\varepsilon) - J(w+\varepsilon))/(2\varepsilon)$ approximates the partial derivative of J with respect to w at the value of w. For sufficiently small ε the approximation is usually accurate to 8 or more significant digits. Greater discrepancies between the results of gradient checking and backpropagation indicate that one or the other is not working properly. As gradient checking is computationally costly, it is only used on a small number of training samples to verify the correctness of backpropagation and disabled for the actual training of the network. Given the more complicated nature of learning in a core as compared to the classical application of backpropagation there are a number of other errors that may occur which prevent learning and are not detected by gradient checking, but the method still serves to eliminate one common source of errors.

Chapter 4

Empirical results

Test results provided here are based on an implementation in Julia[1]. The source code is available on GitHub[2]. As discussed before a thorough quantitative analysis of training results is not within the scope of this project. Instead, several exemplary cases will be used to highlight consistently observed features of the learning process.

The first such program is displayed below in the format, in which it is read by the implementation. To keep things simple and in ASCII-encoding, $\leftarrow$, $\wedge$, and $\neg$ have been written as `<-`, `&` and `-` respectively. The partial program, consisting of clauses 1, 5 and 6, is translated into a core and then trained on samples generated from the full program.

4.1 Comparison of C*- and Ł-interpretation

```
a <- b & -d & -e
b <- a
c <- b
d <- FF
e <- c & d
e <- -a & -b
```

(a) full program

```
a <- b & -d & -e
e <- c & d
e <- -a & -b
```

(b) partial program

Figure 4.1: P1

The first example program illustrates that there are cases in which the method yields good results under C*-interpretation. Training was done with learning rate and momentum of 0.05 under C*-interpretation and 0.02 under Ł-interpretation. During the learning process a number of parameters are measured and reported for intermediate results after every 200 training steps (500 in later examples). *%corr* indicates the percentage of correctly classified training samples, while *eTotal* measures the total number of errors, i.e. the number of incorrect rounded outputs over all output neurons and all samples. In addition, *avgIt* gives the average number of core

[1]julialang.org

[2]github.com/Fr-d-rik/

iterations over all samples and *costJ* is the total value of the error cost function. If the learning algorithm functions correctly, one would expect a steady decline in the cost function, followed by decrease of the total number of errors, which, in turn, leads to an overall rise in the amount of correctly classified samples. Since *%corr* does not differentiate between samples with just one error and samples in which every single neuron is wrongly classified, the latter correlation can be quite weak. Especially when the overall number of errors is still high, *%corr* may even increase, as *eTotal* is reduced, but more evenly distributed across the samples. A similar distribution of errors may also happen in the relationship between cost function and number of errors.

For the C- and C*-interpretation samples, training of the first test program tends to converge after two to three thousand iterations. Depending on the random initialization, usually one of two optima is reached, the first one being at around 80% correct classification, the second one at 100%.

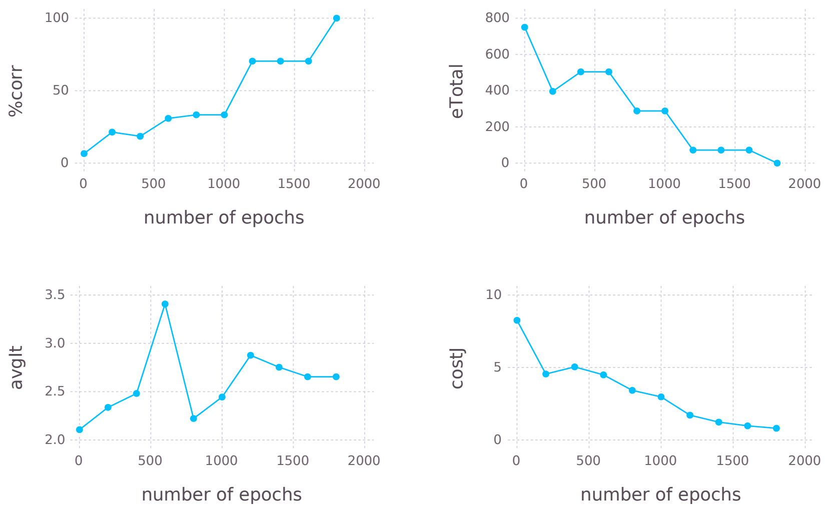

Figure 4.2: P1 training results under C*-interpretation

Under Łukasiewicz interpretation, the cost function can still be seen to generally decrease, though not always monotonically, before it starts to fluctuate. Choosing smaller learning rates remedies the fluctuation to some extent, but in many cases the algorithm does not seem to converge even for small learning rates.[3] The graphs also show less correlation between the cost function and the total number of errors. This can be attributed to the conflict resolution mechanism active under Ł-interpretation, which may generate errors on the final output that are not accounted for in the cost function.

[3]Choosing very small learning rates (in the given example the threshold is at around 0.00003) leads to a steady increase of the cost function. This odd phenomenon has been observed across all examined cases but the source has not been found.

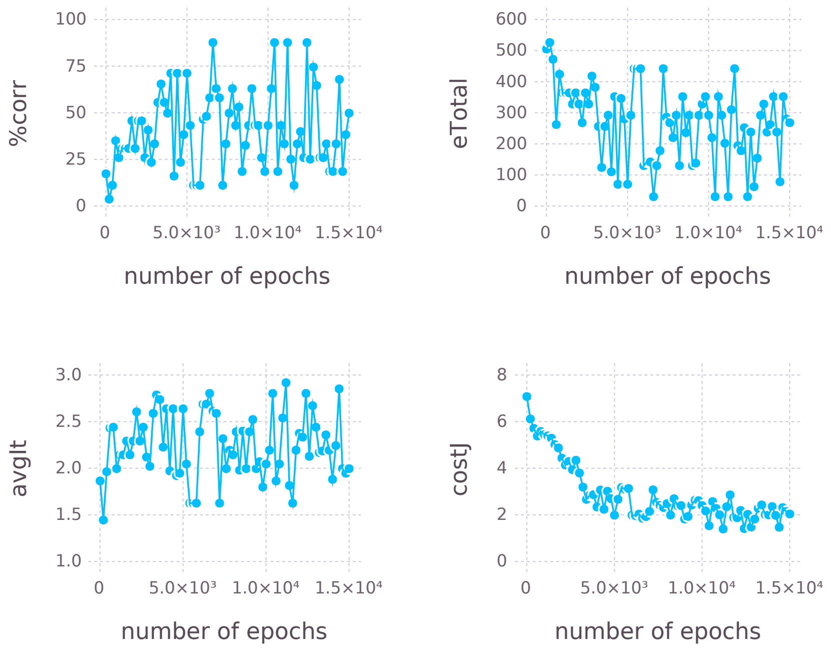

Figure 4.3: P1 training results under Ł-interpretation

4.2 Correlation between error measures

Often times, as displayed by the second example, the learning process quickly gets stuck in local optima under C*-interpretation. While the cost function converges, the average number of iterations keeps fluctuating. Oddly enough, this does not affect the classification performance. Under Ł-interpretation, results are a lot messier, but the correlation between *costJ*, *eTotal* and *%corr* is still clearly recognizable.

```
a <- b & c
b <- d & -e
c <- e
c <- b & -e
d <- a
e <- FF
```

(a) full program

```
a <- b & c
b <- d & -e
c <- e
c <- b & -e
```

(b) partial program

Figure 4.4: P2

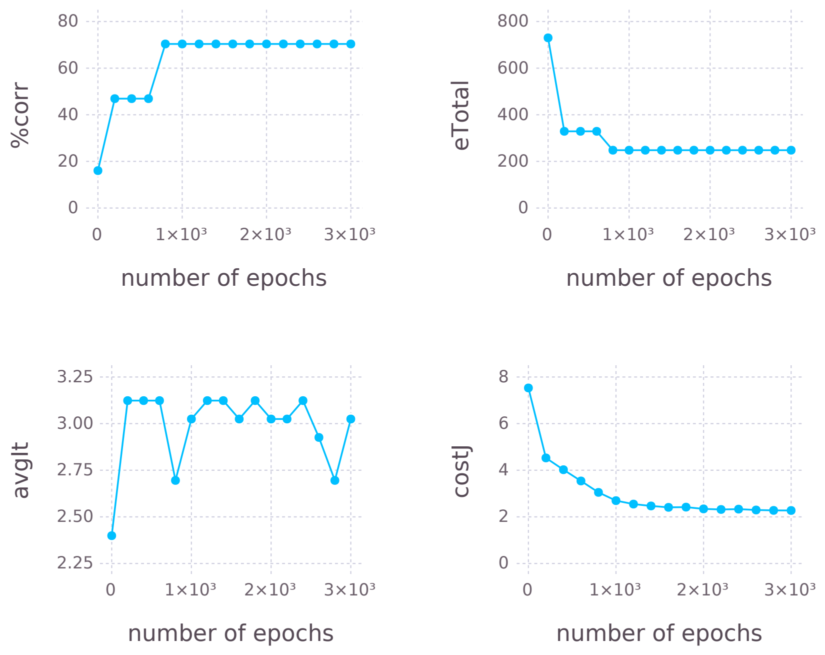

Figure 4.5: P2 training results under C*-interpretation

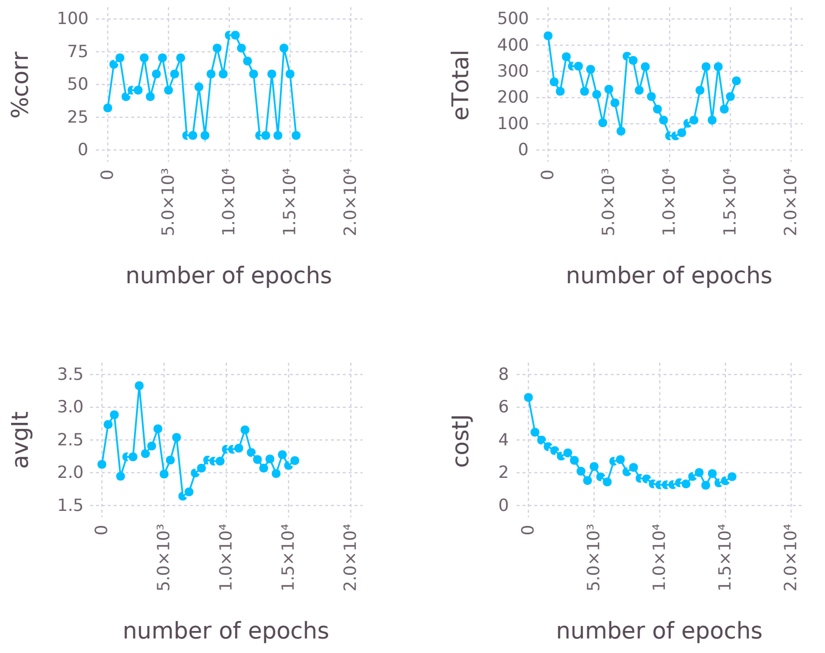

Figure 4.6: P2 training results under Ł-interpretation

4.3 The problem of hidden errors

The third program to be examined contains 8 atoms, three more than the previous programs. This adds 6 neurons to the network and increases the number of training samples from 243 to 6561. The fluctuation in the plots can in part be explained by this fact. Steps of 500 samples make up less than 10% of the total sample size and may at times lead the algorithm in different directions.

What is interesting about this example, is how the algorithm can be observed plummeting in overall performance in the first 500 training steps. This can be attributed to two factors. Under the logistic cost function, which incentivises many smaller errors over fewer large ones, distributing the error may serve to reduce the overall cost, but increase the total amount. Secondly, the backpropagation algorithm is based on the network output. And as the final output is created only after all facts from the input, backpropagation will perceive all misclassifications which are fixed by this final step. Correcting for these errors will decrease the cost function, but does not increase the core's performance. Under Ł-interpretation, the contradiction resolution step contributes to this problem with an additional layer of error correction invisible to the learning algorithm. Moreover, this step cannot be modeled without inhibitory connections, which leaves the algorithm trying - and failing - to correct an error that does not actually exist. There may be multiple reasons for the weak performance under Ł-interpretation, but this is certainly one of them.

In this example, these shortcomings are underlined by the fact that the initial performance from partial background knowledge is much better than the local optimum reached through training.

```
a <- TT
b <- a & -c
d <- c
d <- e
f <- e & -b
f <- a & g & -d
h <- FF
```

(a) full program

```
a <- TT
b <- a & -c
f <- e & -b
f <- a & g & -d
h <- FF
```

(b) partial program

Figure 4.7: P3

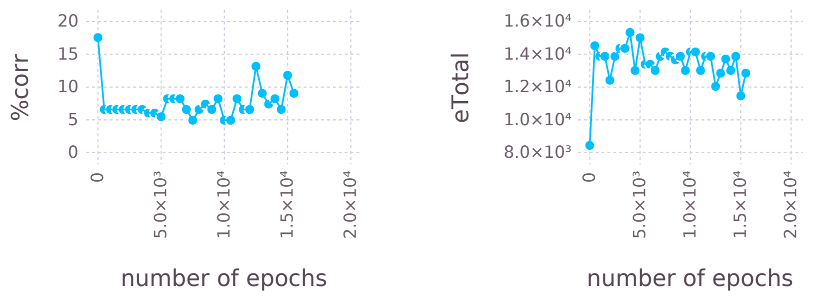

Figure 4.8: P3 training results under C*-interpretation

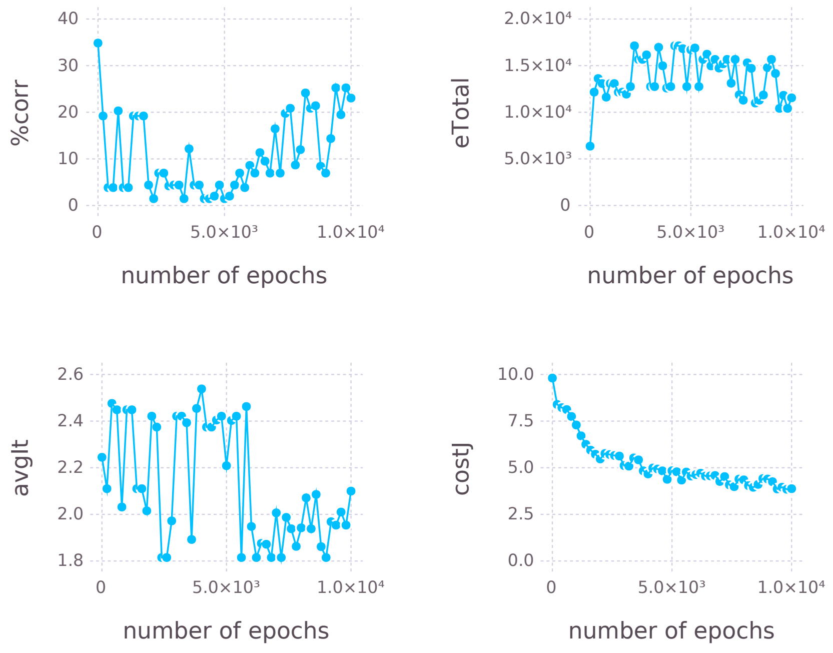

Figure 4.9: P3 training results under Ł-interpretation

4.4 Core compression

The final program examined here, is simply a long chain of inferences. The training results are meant to highlight a phenomenon that is less pronounced but observable in most trained cores. In this extreme case the partial program starts with an average number of iterations of 4.14, which immediately drops to around 2, changing very little thereafter. This number includes the last iteration where input and output must be equal. Therefore, in all but very few cases, the inference is compressed into one iteration. In general, trained cores tend to have fewer iterations on average than their translated counterparts. This can be explained by the fact, that the backpropagation algorithm does not take multiple iterations into account and optimizes for correct output after just one iteration. This property does not decrease the performance of trained cores, but it does suggest that they do not fully utilize the core-architecture.

```
b <- a
c <- b
d <- c
e <- d
f <- e
g <- f
g <- FF
```

(a) full program

```
b <- a
c <- b
d <- c
e <- d
f <- e
g <- FF
```

(b) partial program

Figure 4.10: P4

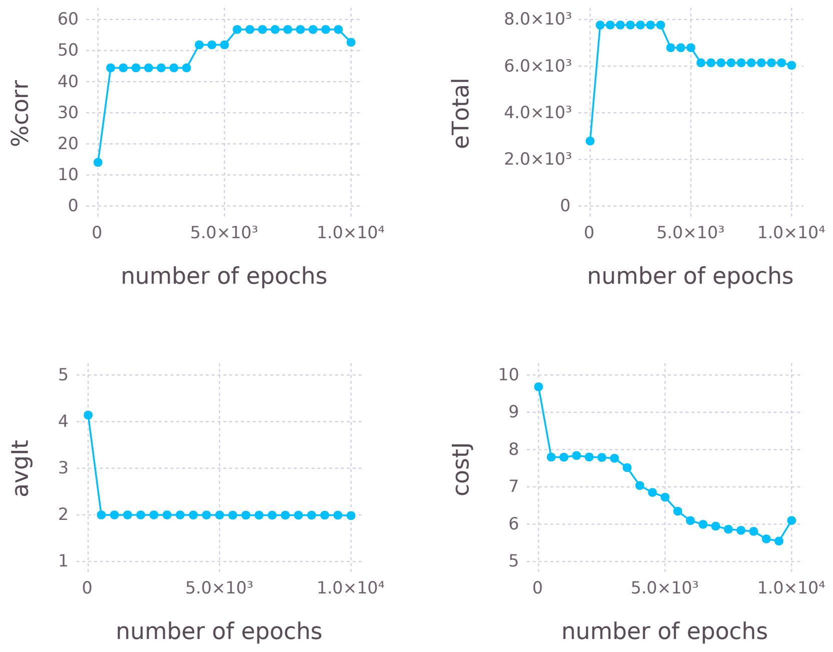

Figure 4.11: P4 training results under C*-interpretation

4.5 Analysis through rule extraction

The developed rule extraction method is not yet suited to provide a complete picture of the information contained in a core, but it may be used to take a look at individual neurons and their activation rules. This can be done both for translated and trained cores to see, what differences remain after training.

In the third example above, for instance, in which the core was generated from the program P3 with two missing clauses `d <- c` and `d <- e` and then trained, it turns out that the $\mathtt{d}^\top$ neuron's activation rules in the trained core match the full program. While learning was successful with regard to $\mathtt{d}^\top$, $\mathtt{d}^\perp$ does not show any activation in the trained core, whereas the core containing the complete translated program has an activation in $\mathtt{d}^\perp$ when both $\mathtt{c}^\perp$ and $\mathtt{e}^\perp$ are active.

These findings alone do not explain, why the trained core classifies less than 10% of samples correctly. Looking further, it can be found that other inference structures have largely broken down. For instance, $\mathtt{f}^\top$ has three activation rules in the core containing the complete program. In the trained core, two rules are extracted, one of which is wrong. This does not mean, that the connections to the $\mathtt{f}^\top$ output neuron are necessarily wrong. Due to the core's multiple iterations, the activation patterns of different neurons are highly interdependent and errors are hard to localize.

out($\mathtt{d}^\top$)	$\mathtt{a}^\top$	$\mathtt{a}^\perp$	$\mathtt{b}^\top$	$\mathtt{b}^\perp$	$\mathtt{c}^\top$	$\mathtt{c}^\perp$	$\mathtt{d}^\top$	$\mathtt{d}^\perp$	$\mathtt{e}^\top$	$\mathtt{e}^\perp$	$\mathtt{f}^\top$	$\mathtt{f}^\perp$	$\mathtt{g}^\top$	$\mathtt{g}^\perp$	$\mathtt{h}^\top$	$\mathtt{h}^\perp$
translated 1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
translated 2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
translated 3	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
trained 1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
trained 2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
trained 3	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Figure 4.12: Extracted minimal activating inputs of $\mathtt{d}^\top$ in P3

out($\mathtt{f}^\top$)	$\mathtt{a}^\top$	$\mathtt{a}^\perp$	$\mathtt{b}^\top$	$\mathtt{b}^\perp$	$\mathtt{c}^\top$	$\mathtt{c}^\perp$	$\mathtt{d}^\top$	$\mathtt{d}^\perp$	$\mathtt{e}^\top$	$\mathtt{e}^\perp$	$\mathtt{f}^\top$	$\mathtt{f}^\perp$	$\mathtt{g}^\top$	$\mathtt{g}^\perp$	$\mathtt{h}^\top$	$\mathtt{h}^\perp$
translated 1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
translated 2	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
translated 3	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0
trained 1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
trained 2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Figure 4.13: Extracted minimal activating inputs of $\mathtt{f}^\top$ in P3

Chapter 5

Discussion and outlook

The goal of this project was to investigate the applicability of the backpropagation algorithm to Łukasiewicz logic cores and this goal has been met, be it with mixed results. In order to train the cores, a number of modifications had to be made. Sigmoidal activation units were introduced, along with the proposition of unipolar weights, and several options for the generation of training data have been discussed. These steps have been motivated and provided with formal proofs where it was considered necessary, resulting in a trainable core as an intermediate result. Further research on the topic, for example using other supervised learning methods, can be based on this type of core as well. The training process itself is more problematic, as training of cores under Ł-interpretation clearly does not perform as well as desired. Whether this weak performance can be sufficiently improved by standard augmentations of the algorithm or whether the contradiction resolution mechanism used in Ł-interpretation prevents backpropagation from functioning properly on a more general level, is not clear at the moment. In addition, a rule extraction algorithm has been proposed for cores trained under C- or C*-interpretation. This is very much a work in progress, but should also in its current state provide a starting off point for other ventures in that direction. In the following, a number of problems of the project are discussed, whose resolution would offer ways of tackling the issue of weak performance. They are followed by a list of more specific objectives for future endeavors to improve the project.

5.1 Challenges

Proof for last-iteration backpropagation

Possibly a very problematic aspect of this work is its failure to provide a proof for the functioning of backpropagation in the way, in which it has been applied. While it is an established fact, that an MLP with a sufficient number of hidden units will be able to model the behaviour of the types of logic programs presented here, and it is evident that a unipolar MLP embedded in the core structure is powerful enough to accomplish this task as well, it is less clear how a unipolar core can be trained to reach this performance. The method of training the core only on the last iteration, in which the input equals the output, does not take the core's capacity for multiple iterations into account. It also means that the algorithm is only guaranteed to work as intended on those inputs which are themselves fixed points. One indication that backpropagation may not be the method of choice for training cores is the empirical result, that the average number of iterations in the core tends to decline rapidly over the learning process, before usually settling somewhere around 2, which is also the minimum possible number for all non-fixpoint inputs. This suggests that information about the program is compressed in the network, leading to redundancy, rather than building on itself, as seen in an untrained core.

Missing link between C* and Ł

Another problem which remains unsolved in this project is the disparity between Ł- and C*-interpretation. As the empirical results indicate, Ł-interpretation is a lot harder to train and the C- and C*-interpretations have been used, in large part, as a stand-in, so learning and rule extraction could be explored, in the hope that the gap to Ł-interpretation could be bridged later on. In case this problem cannot be addressed in a satisfying manner, another route would be to explore, how much training a network on C*-interpretation samples can improve performance on Ł-interpretation test sets. While C*-models are not generally equal to Ł-models, their similarities might be sufficient to motivate learning on one interpretation in order to improve performance on the other.

Implausibility of C*-samples

The problem with this last option and learning on C*-samples in general is that, while it is easy to produce C*-interpretation samples for training in the course of this project, any actual application scenario for Łukasiewicz cores will naturally provide Ł-model samples. A method for translating them into their C*-model equivalents has not been put forward and may well not exist.

5.2 Improvements

Modified backpropagation for better results

As has been outlined in the introductory section, there are a great number of modifications that may be applied to the backpropagation algorithm in order to boost its performance. The version of on-line backpropagation with momentum and standard gradient descent used here is sufficient to provide qualitative results, i.e. whether or not the system improves performance through learning at all. A quantitative analysis of how well a core can be trained would be the consistent next step. Such an analysis should employ some additions to backpropagation likely including a better initialization method for the weights of added neurons and running of multiple initializations. Performance can then be measured by standard means of crossvalidation, partitioning the samples in training, test, and validation set, to compare the results to other approaches with a sample size that seems plausible for application scenarios.

Exploration of other optimization methods

Backpropagation only being one of many optimization algorithm for neural networks, it is also promising to look at other more general methods. Most problems with the current implementation likely stem from the tight focus of backpropagation on the neural network, which fails to take the rest of the core architecture into account. Genetic algorithms [17] are likely a good fit, as they grant more freedom in defining fitness criteria. These could, for example, incentivise a higher number of iterations, preventing the network from condensing all information into one iteration.

Completion of the extraction algorithm

The proposed extraction algorithm is another area where incremental progress may be achieved. In its current state the method is not finished, because it is missing a translation of discovered rules into actual logic program clauses. In trained cores with less than perfect classification this task bears some challenges, as they may not represent a clean logic program. The relative values of completeness and soundness must then be weighed against each other in constructing translation methods, a task which warrants its own thoughtful investigation.

A further goal is to adapt the algorithm to proper Łukasiewicz models. However, doing so may turn out to be so inefficient, due to the loss of monotonicity, that other extraction algorithms should be considered instead. A property which may still prove useful for pruning is the monotonicity of Ł-models with regard to the addition of negative facts[1].

5.3 Closing remarks

Several problems remain to be solved before the neural-symbolic cycle for three-valued Łukasiewicz cores can be fully closed. In conclusion, I hope that the exploration performed in this thesis will provide reference points for future work on this topic. Work which will contribute to our general understanding, of how symbolic and statistical systems interact and hopefully lead to a point, where neural-symbolic methods can be employed to tackle those problems, that are difficult to address with either paradigm exclusively.

[1] Proof sketch: Adding a negative $A \leftarrow \bot$ fact either changes the value of A from u to $\bot$ or not at all. For each atom B: If $B = \top$, then there is a clause $B \leftarrow \mathit{body}$ with $\mathit{body} = \top$ and as body is a conjunction of literals it cannot have contained A with value u. If $B = \bot$, then for all clauses $B \leftarrow \mathit{body}$, $\mathit{body} = \bot$. There is at least one literal $L = \bot$ in each body and the values of the other conjuncts do not matter. Therefore each atom which had a value other than u before the addition of $A \leftarrow \bot$ will retain that value.

Appendix A

Proofs

Backpropagation for logistic cost function with unipolar weights

To follow the notation used by [18], a few terms have to be introduced.

Logistic cost function: $J(\vec{w}) = \frac{1}{m}\sum_{i=1}^{m}[(y^{(i)}\log h_{\vec{w}}(x^{(i)}) + (1 - y^{(i)})\log(1 - h_{\vec{w}}(x^{(i)}))]$

Error for one sample d**:** $E_d(\vec{w}) = \sum_{k\in outputs}[(t_k \log o_k) + (1 - t_k)\log(1 - o_k)]$

Weight update term: $\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}}$

Weighted sum of inputs to j**:** $net_j = x_0 \cdot w_0 + \sum_{k>0} x_{jk} \cdot \frac{1}{2}(w_{jk})^2$

Sigmoid function: $\sigma(x) = 1/(1 + e^{-x})$

Error term delta: $\delta_j = -\frac{\partial E_d}{\partial net_j}$

The objective of the backpropagation algorithm is to produce weight updates Δw_{ji} for all weights for connections from i to j in the network. The learning rate η will be set independently and therefore the following proof is mostly concerned with a derivation of $\frac{\partial E_d}{\partial w_{ji}}$ for each weight.

Because the weight w_{ji} always enters into E_d in the context of net_j, using the chain rule, the partial derivative can be rewritten as follows:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

The second factor in the right term yields:

$$\frac{\partial net_j}{\partial w_{ji}} = \begin{cases} i = 0 : x_{ji} \\ i > 0 : w_{ji} \cdot x_{ji} \end{cases}$$

Using above definition, the formula for weight updates follows.

$$\Delta w_{ji} = -\eta\frac{\partial Ed}{\partial w_{ji}} = \begin{cases} i = 0 : \eta \cdot \delta_j \cdot x_{ji} \\ i > 0 : \eta \cdot \delta_j \cdot w_{ji} \cdot x_{ji} \end{cases}$$

The way in which δ_j is derived depends on the layer, in which neuron j is located.

δ_j for output layer units

As net_j only affects output o_j, another expansion yields

$$\frac{\partial Ed}{\partial net_j} = \frac{\partial Ed}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

The two resulting derivatives can be simplified.

$$\frac{\partial Ed}{\partial o_j} = \frac{t_j - o_j}{o_j \cdot (1 - o_j)}$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j \cdot (1 - o_j)$$

This yields the result.

$$\delta_j = -\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)$$

δ_j for hidden layer units

net_j affects E_d only through first o_j and then the weighted inputs net_k of all neurons that j has outgoing connections to, i.e. that are *downstream* from j (noted as $k \in DS(j)$ below). With this knowledge the following expansions can be made.

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in DS(j)} \frac{\partial E_d}{\partial net_k} \cdot \frac{\partial net_k}{\partial net_j} = \sum_{k \in DS(j)} -\delta_k \cdot \frac{\partial net_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}$$

Both partial derivatives in the right term can be transformed. It helps to know that the bias-unit has no incoming connection and thus $j > 0$.

$$\frac{\partial net_k}{\partial o_j} = \frac{\partial net_k}{\partial x_{kj}} = \frac{\partial}{\partial x_{kj}}(w_{k0} \cdot x_{k0} \cdot \sum_{j>0} \frac{1}{2}(w_{kj})^2 \cdot x_{kj}) \overset{j>0}{=} \frac{1}{2}(w_{kj})^2$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\sigma(net_j)}{\partial net_j} = \sigma' = o_j \cdot (1 - o_j)$$

The result follows.

$$\delta_j = o_j \cdot (1 - o_j) \cdot \sum_{k \in DS(j)} \delta_k \cdot \frac{1}{2}(w_{kj})^2$$

Preservation of Łukasiewicz semantics in sigmoidal cores

Let $[0, A^-]$ and $[A^+, 1]$ be two disjoint intervals representing firing and non-firing activation respectively. Assume $sig(z) = 1/(1 + e^{-(z)})$, $\omega > 0$ and $deg \geq 1$, the last being the maximum indegree among neurons of the hidden/output layer.

In the following notation, activations A indicate whether they represent firing or not firing $(+,-)$, are in hidden or output layer (h,o) and associate with a *true* or *false* neuron $(\top,\bot)$, where needed. Note that activations in the input layer are discrete and therefore written as 0 and 1.

Proper functioning of logic gates in the network is guaranteed, as long as the following inequalities hold. For each layer and logic gate type there is one formula specifying the minimal input which must lead to activation and one formula for the maximal possible input that must not activate the gate.

- Logic gates in the hidden layer:
  - On $\top$ clause neurons ('and'-gate)
    $\min(A^+_{h\top}) = sig(deg \cdot 1 \cdot \omega - (deg \cdot \omega - \frac{\omega}{2})) \geq A^+$
    $\max(A^-_{h\top}) = sig((deg - 1) \cdot 1 \cdot \omega + 0 \cdot \omega - (deg \cdot \omega - \frac{\omega}{2})) \leq A^-$
  - On $\bot$ clause neurons ('or'-gate)
    $\min(A^+_{h\bot}) = sig(1 \cdot \omega - \frac{\omega}{2}) \geq A^+$
    $\max(A^-_{h\bot}) = sig(0 \cdot \omega - \frac{\omega}{2}) \leq A^-$
- Logic gates in the output layer:
  - On $\top$ clause neurons ('or'-gate)
    $\min(A^+_{o\top}) = sig(\min(A^+_h) \cdot \omega - \frac{\omega}{2}) \geq A^+$
    $\max(A^-_{o\top}) = sig(deg \cdot \max(A^-_h) - \frac{\omega}{2}) \leq A^-$
  - On $\bot$ clause neurons ('and'-gate)
    $\min(A^+_{o\bot}) = sig(deg \cdot \min(A^+_h) \cdot \omega - (deg \cdot \omega - \frac{\omega}{2})) \geq A^+$
    $\max(A^-_{o\bot}) = sig((deg - 1) \cdot 1 \cdot \omega + \max(A^-_h) \cdot \omega - (deg \cdot \omega - \frac{\omega}{2})) \leq A^-$

The hidden layer formulas reduce to:

$$\begin{aligned}
&\min(A^+_{h\top}) = sig(\tfrac{\omega}{2}) \geq A^+ \\
&\max(A^-_{h\top}) = sig(-\tfrac{\omega}{2})) \leq A^- \\
&\min(A^+_{h\bot}) = sig(\tfrac{\omega}{2}) \geq A^+ \\
&\max(A^-_{h\bot}) = sig(-\tfrac{\omega}{2}) \leq A^-
\end{aligned}$$

The distinction between $h\top$ and $h\bot$ can thus be ignored in the following. Also it is clear now that $A^+ > \frac{1}{2} > A^-$. Inequalities regarding the output layer can be transformed into:

1 $\min(A^+_{o\top}) = sig((\min(A^+_h) - \frac{1}{2}) \cdot \omega) \geq A^+$
2 $\max(A^-_{o\top}) = sig((deg \cdot \max(A^-_h) - \frac{1}{2}) \cdot \omega) \leq A^-$
3 $\min(A^+_{o\bot}) = sig((deg \cdot \min(A^+_h) - \frac{2deg-1}{2}) \cdot \omega) \geq A^+$
4 $\max(A^-_{o\bot}) = sig((\max(A^-_h) - \frac{1}{2}) \cdot \omega) \leq A^-$

It can be seen that formulas 1 and 4 imply the hidden layer inequalities, 2 implies 4 and 3 implies 1[1]. Satisfying formulas 2 and 3 therefore satisfies the other inequalities as well. Now, as it has been established that:

[1] $(deg \cdot \min(A^+_h) - \frac{2deg-1}{2})$ is maximal with $deg = 1$, then equal to $(\min(A^+_h) - \frac{1}{2})$

$\min(A_h^+) = sig(\frac{\omega}{2})$
$\max(A_h^-) = sig(-\frac{\omega}{2})$

In the 3 layer network it follows:

$\max(A_{o\top}^-) = sig((deg \cdot sig(-\frac{\omega}{2}) - \frac{1}{2}) \cdot \omega)$
$\min(A_{o\bot}^+) = sig((deg \cdot sig(\frac{\omega}{2}) - \frac{2deg-1}{2}) \cdot \omega)$

This results in the final inequalities

$sig((deg \cdot sig(-\frac{\omega}{2}) - \frac{1}{2}) \cdot \omega) \leq A^-$
$sig((deg \cdot sig(\frac{\omega}{2}) - \frac{2deg-1}{2}) \cdot \omega) \geq A^+$

Knowing that $A^+ > \frac{1}{2} > A^-$, these have a solution precisely iff

$(deg \cdot sig(-\frac{\omega}{2}) - \frac{1}{2}) < 0$
$(deg \cdot sig(\frac{\omega}{2}) - \frac{2deg-1}{2}) > 0$

Either is true iff $\omega > 2\log(2deg - 1)$.

Relation of C*-models and Ł-models

Claim: If an initial activation under C*-interpretation leads to a fixed point with no contradiction, the same fixed point will be reached under Ł-interpretation. Assuming the core architecture implementation itself is correct, this means that all least C*-models are also least Ł-models.

Proof sketch:

(1) By design, the one thing differentiating C*-consequence from Ł-consequence is that in Ł contradictions are resolved at the end of each iteration.
(2) Due to monotonicity, every output neuron activated while finding a C* fixed point stays active in further iterations.
(3) It follows from (1), that if the fixed point of C* and Ł on the same input differ, it's because a contradiction was resolved in Ł.
(4) It follows from (2), that if a contradiction occurs in C during fixed point generation, it will still be there in the fixed point.
(5) (3) and (4) imply, that if there is no contradiction in the C* fixed point, none was resolved in the corresponding Ł fixed point and thus, the fixed points are equal.

Appendix B

Extraction algorithm

Type definitions

```
Type **Vertex**
lbub        /* true if lower boundary, false if upper boundary */
target      /* index of target output neuron */
isRule      /* true if rule or subsumed by a rule */
isPos       /* true if target is active given activation in val */
decID       /* decimal ID */
val         /* array of input values */
mem         /* memorizes, which successors should be generated */
```

Algorithm 1: Vertex

```
Type RuleSet
lbRules     /* list of found minimal activating inputs */
ubRules     /* list of found maximal non-activating inputs */
```

Algorithm 2: RuleSet

Auxiliary functions

```
Function get_id     /* produces key to identify vertices */
Input: val
Output: decimal value of val read as a binary number
```

Algorithm 3: get_id

```
Function succ_id        /* computes decID that differs from vertex at index */
Input: vertex, index
if vertex.mem[index] = 0 then
    newID := -1
else
    max := length(vertex.val)
    newID := vertex.decID
    newID := newID + $2^{(max-index)}$ if vertex.lbub , newID - $2^{(max-index)}$ otherwise
Output: newID
```

Algorithm 4: succ_id

```
Function make_succ                /* create succcessor from copy of vertex */
Input: vertex,index
newV := deepcopy(vertex)
newV.val[index] := 1 if vertex.lbub, 0 otherwise
newV.mem := abs(vertex.mem)             /* 'forget' temporary -1 blockings */
newV.mem[index] := 0
newV.decID = succ_id(vertex,index)
Output: newV
```

Algorithm 5: make_succ

```
Function in_rules_lb    /* true if vertex subsumed by rule in lower bound */
Input: vertex, ruleset

for i = 1:length(ruleset.lbRules) do
    if minimum(vertex.val - ruleset.lbRules[i]) == 0 then
        isSub = true
        subRule = ruleset.lbRules[i]             /* also returns subsuming rule */
        Output: isSub, subRule
Output: false, zeros

Function in_rules_ub                 /* analogous function for upper bound */
```

Algorithm 6: in_rules_lb, in_rules_ub

Central algorithm

```
Function test                              /* finds rules and subsumed anti-rules */
Input: core, vertex, ruleset
cOut := run_core(core, vertex.val)                              /* get core output */
vertex.isPos := true if cOut[vertex.target] = 1, false otherwise
if vertex.lbub then
    if vertex.isPos then
        ruleset.lbRules ← vertex.val
        vertex.mem := zeros
        vertex.isRule := true
                                                          /* lower bound rule found */
    else
        isSub, subRule := in_rules_ub(vertex, ruleset)
        if isSub then set vertex.mem to -1 where subRule = 1 and mem = 1
                                        /* isSub: subsumed by lower bound anti-rule */
else
    if vertex.isPos then
        isSub, subRule = in_rules_lb(vertex, ruleset)
        if isSub then set vertex.mem to -1 where subRule = 0 and mem = 1
                                        /* isSub: subsumed by upper bound anti-rule */
    else
        ruleset.ubRules ← vertex.val
        vertex.mem := zeros
        vertex.isRule := true
                                                          /* upper bound rule found */
```

Algorithm 7: test

```
Function successors
Input: core,vertex, vvector, queue
test(core, vertex, ruleset)
noMatchSuccc := true if $minimum(vertex.mem) \geq 0$, false otherwise
for $i$ = 1:length(vertex.mem) do
    if $vertex.mem[i] = 1$ then                        /* traverse valid successors */
        succID := succ_id(vertex,i)
        if ¬isdefined(vvector, succID) then              /* create if necessary */
            vvector[succID] := make_succ(vertex, i)
            test(core, vvector[succID], ruleset)
            queue ← succID
        if vvector[succID].isRule then vertex.mem[i] := 0   /* pool successor info */
        if vvector[succID].isPos = vertex.isPos then noMatchSucc := false
if noMatchSucc then                                     /* test for anti-rules */
    if vertex.lbub ∧ ¬vertex.isPos then ruleset.ubRules ← vertex.val
    if ¬vertex.lbub ∧ vertex.isPos then ruleset.lbRules ← vertex.val
for $i$ = 1:length(vertex.mem) do          /* distribute info to all successors */
    if $vertex.mem[i] = 1$ then
        succID := succ_id(vertex, i)
        set vvector[succID].mem to 0 where vertex.mem is 0
```

Algorithm 8: successors

```
Function rules_for_target
Input: core, target
⊥ := vertex for bottom element for core
⊤ := vertex for top element for core
queue := Queue(integers)                    /* stores keys of vertices such that */
queue ← ⊥.decID,⊤.decID                     /* the boundaries alternate by layer */
vvector := vector(vertices) with $3^n$ entries, where n = number of output units in core
vvector[begin] := ⊥
vvector[end] := ⊤
ruleset := empty RuleSet
while ¬empty(queue) do            /* repeat until all vertices pruned or covered */
    index := dequeue(queue)
    v := vvector[index]
    successors(core, v, ruleset, vvector, queue
```

Algorithm 9: rules_for_target

List of Figures

List of Algorithms

Bibliography

[1] S. Amato et al. Simulated annealing approach in backpropagation *Neurocomputing (Volume 3, Issues 5–6) pp. 207–220.* 1991

[2] S. Bader, P. Hitzler. Dimensions of neural-symbolic integration - a structured survey. *In: S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb and J. Woods (eds). We Will Show Them: Essays in Honour of Dov Gabbay, Volume 1. International Federation for Computational Logic, College Publications, pp. 167-194.* 2005

[3] S. Bader, P. Hitzler, S. Hölldobler, A. Witzel. A fully connectionist model generator for covered first-order logic programs. *In: Manuela M. Veloso, ed. Proceedings of the Twentieth International Joint Conference on Artificial Intelligence, pp. 666–671, Menlo Park CA, January 2007. AAAI Press.* 2007

[4] C. Berger, B. Rumpe. Autonomous Driving-5 Years after the Urban Challenge : The Anticipatory Vehicle as a Cyber-Physical System *In: 2012 proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012): Essen, Germany, 3 - 7 September 2012, pp. 789-798.* 2012

[5] R. Byrne. Suppressing valid inferences with conditionals. *Cognition, 31, pp. 61-83.* 1989

[6] E.-A. Dietz, S. Hölldobler, M. Ragni. A Computational Logic Approach to the Suppression Task *In: N. Miyake D. Peebles and R. P. Cooper, eds., Proceedings of the 34th Annual Conference of the Cognitive Science Society, 1500-1505.* 2012

[7] D. Erdogmus et al. Linear-least-squares initialization of multilayer perceptrons through backpropagation of the desired response *IEEE Transactions on Neural Networks (Volume 16, Issue 2) pp. 325-337.* 2005

[8] A.S. d'Avila Garcez, K. Broda, D.M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach *Artificial Intelligence 125, pp. 155–207* 2001

[9] A.S. d'Avila Garcez, K. Broda, D.M. Gabbay. Neural-Symbolic Learning Systems:Foundations and Applications. *Springer. ISBN 9781447102113.* 2002

[10] A. d'Avila Garcez, T. R. Besold, L. de Raedt, P. Földiak, P. Hitzler, T. Icard, K.-U. Kühnberger, L. Lamb, R. Miikkulainen, D. Silver. Neural-Symbolic Learning and Reasoning: Contributions and Challenges. *In: AAAI Technical Report of the AAAI Spring 2015*

Symposium on Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches, SS-15-03, pp. 18-21. AAAI Press. 2015

[11] H. Gust, K.-U. Kühnberger, P. Geibel. Learning models of predicate logical theories with neural networks based on topos theory. *In: B. Hammer, P. Hitzler, eds. Perspectives of Neural-Symbolic Integration, pp. 233-264. Springer. ISBN 9783540739548.* 2007

[12] S. Hölldobler, Y. Kalinke. Towards a massively parallel computational model for logic programming. *In: Proceedings ECAI94 Workshop on Combining Symbolic and Connectionist Processing, pp. 68–77.* 1994

[13] S. Hölldobler and C. D. P. Kencana Ramli. Logics and Networks for Human Reasoning *Artificial Neural Networks – ICANN 2009, Lecture Notes in Computer Science Volume 5769, pp. 85-94.* 2009

[14] F. Hsu. Behind Deep Blue: Building the Computer that Defeated the World Chess Champion *Princeton University Press. ISBN 0691090653.* 2002

[15] J. Łukasiewicz. Treść wykładu pożegnalnego wygłoszonego w auli Uniwersytetu Warszawskiego 7 marca 1918 r. *Pro arte et studio 3, pp. 3–4.* 1918 (English translation: Farewell lecture delivered in the Warsaw University Lecture Hall on March 7, *In: L. Borkowski, ed. Jan Lukasiewicz Selected Works. North Holland, pp. 87-88.* 1990)

[16] J. Łukasiewicz. O logice trójwartościowej *Ruch Filozoficzny, 5, pp. 169-171.* 1920 (English translation: On Three-Valued Logic. *In: L. Borkowski, ed. Jan Lukasiewicz Selected Works. North Holland, pp. 87-88.* 1990)

[17] M. Mitchell. An Introduction to Genetic Algorithms. *Cambridge, MA: MIT Press. ISBN 9780585030944.* 1996

[18] T. Mitchell. Machine Learning, *McGraw Hill. ISBN 0070428077.* 1997

[19] V. Mnih et al. Human-level control through deep reinforcement learning *Nature 518, pp. 529–533.* 2015

[20] R. Plamondon, S. N. Srihari. Online and off-line handwriting recognition: a comprehensive survey *IEEE Transactions on Pattern Analysis and Machine Intelligence (Volume 22, Issue 1) pp. 63-84.* 2000

[21] J. A. Robinson, A. Voronkov (eds). Handbook of Automated Reasoning *MIT Press. ISBN 0444508139.* 2001

[22] D. E. Rumelhart, G. E. Hinton, R. J. Williams. Learning internal representations by error propagation. *In: D. E. Rumelhart, J. L. McClelland and the PDP Research Group (eds), Parallel Distributed Processing, vol. 1: Foundations, pp. 318–362. MIT Press.* 1986

[23] K. Stenning, M. van Lambalgen. Human reasoning and cognitive science. *MIT Press. ISBN: 9780262195836.* 2008

[24] T. Winograd. Understanding Natural Language *Academic Press. ISBN 0127597506.* 1972

Declaration of Authorship