Neural-Symbolic Integration

Dissertation

zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr. rer. nat)

vorgelegt an der

Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Sebastian Bader
geb. am 22. Mai 1977 in Rostock

Dresden, Oktober 2009

Gutachter:	Prof. Dr. rer. nat. habil Steffen Hölldobler (Technische Universität Dresden) Prof. Dr. rer. nat. habil Barbara Hammer (Technische Universität Clausthal)
Verteidigung:	5. Oktober 2009

Neural-Symbolic Integration

Sebastian Bader

Dresden, October 2009

To my family.

Contents

List of Figures

Preface & Acknowledgement

My interest in the area of neural-symbolic integration started when I was an undergraduate student at TU Dresden. In all classes on artificial neural networks that I took, some questions remained open, among them were the following two which are key problems tackled in this thesis: "What did the network really learn?" and "How can I help the network using background knowledge?" While being a student in the *"International Masters Program in Computational Logic"*, I was given the chance to dive deeper into this fascinating area. First, during my stay in Brisbane, Australia, and then during my master's thesis. Finally, I ended up doing my PhD on this subject.

During the last years I met a number of interesting people which contributed to this thesis in many ways: by spending time to discuss the subject and by just being there to answer my numerous questions.

Thank you all!

In particular I would like to thank Prof. Steffen Hölldobler for being my supervisor, for heading the *"International Masters Program in Computational Logic"* and for being the head of "my" research group in Dresden. I would also like to thank Dr. Pascal Hitzler for supervising me and for being available for discussions at any time. And I am thankful to Prof. Heiko Vogler for his work as head of the DFG graduate programme 334 *"Specification of discrete processes and systems of processes by operational models and logics"*. But of course there have been many others. And, I would like to thank my family for all their support and love.

This thesis has been written with financial support from the DFG graduate programme 334 "Specification of discrete processes and systems of processes by operational models and logics".

1 Introduction and Motivation

... where we discuss reasons to study neural-symbolic integration, review related work and introduce a new classification scheme. Then, a number of challenging problems are mentioned which have been identified to be the main questions that need to be answered. Finally, we discuss the structure of this thesis.

This chapter is partly based on [BH05] where we presented the classification scheme, and on [BHH04] where we discussed the challenge problems.

1.1 Motivation for the Study of Neural-Symbolic Integration

To motivate the study of neural-symbolic integration, we discuss it from different points of view:

- from an *artificial intelligence* perspective,
- from a *neuroscience* perspective, and
- from a *cognitive science* perspective.

Afterwards, we present the so-called *neural-symbolic cycle*, which serves as a general guideline to create integrated systems.

1.1.1 An Artificial Intelligence Perspective

One of the ultimate goals of artificial intelligence is the creation of agents with human-like intelligence. This has been tried to achieve using various approaches. Among the most prominent are logic-based symbolic systems and artificial neural networks. Both paradigms have quite controversial advantages and disadvantages. Researchers in the field of *neural-symbolic integration* try to bridge the gap between them. The main goal for this line of research is the development of systems that conjoin the advantages of both.

Connectionist systems – also called artificial neural networks – are a powerful approach for machine learning, inspired by biology and neurology. They are trainable from raw data, even if the data is noisy and inconsistent and thus well suited to adapt to new situations. They are furthermore robust in sense that they degrade gracefully. Even if parts of the system fail, it is still working. Unfortunately, they do not possess a declarative semantics and have problems while handling structured data. Available (symbolic) background knowledge which exists in many application domains can hardly be used together with such a system. Nevertheless, connectionist systems have been used in many applications.

Symbolic systems on the other hand are usually based on some kind of logic. They possess a declarative semantics and knowledge can be modelled in a human-like fashion. Thus we can easily use existing knowledge and it is easy to handle structured objects. Unfortunately, those systems are hard to refine from real world data which usually is noisy and they are hard to design if no expert knowledge is available. As well as connectionist systems, they have been successfully used in many applications. To conjoin the advantages of both approaches we would like to design an integrated system

- in which knowledge can be explicitly stated in a human-writable form,
- which is trainable from noisy and inconsistent data, and
- from which refined knowledge can be extracted into a human-readable form.

To achieve this goal, we need to solve a couple of open problems. The main problems can be summarised as follows:

- How can symbolic knowledge be expressed within a connectionist system?
- How can symbolic knowledge be used while training a connectionist system?
- How can symbolic knowledge be extracted from a connectionist system?

These are three of the nine research problems which are discussed in Section 1.4.

1.1.2 A Neuroscience Perspective

Most people agree that humans are capable of manipulating symbols on a neural substrate. Using our brain as hardware we can draw symbolic conclusions using previously acquired knowledge. How this is achieved is completely unclear yet, we just know that we can do it. Probably not being the driving force behind the study of neural-symbolic integration, the field of (computational) neuroscience could still profit from results achieved here, and vice versa.

One goal of neural-symbolic integration is the realisation of symbolic processes within artificial neural networks. Even though the networks usually used in this area are only coarse abstractions of our brain, they behave in some situations at least similar. Assuming we manage to implement a symbolic computation within such an abstracted network, neuroscientist might apply the underlying ideas to biologically more plausible models as well. Eventually, this might help to understand the higher cognitive processes performed in our brain. Because we know that our brain is able to manipulate symbols, we might hope that insights from neuroscience might also help to develop better neural-symbolic systems.

1.1.3 A Cognitive Science Perspective

Researchers in the interdisciplinary field of cognitive science study the nature of intelligence and the brain. As mentioned above, humans are able to process symbolic knowledge using their embedded neural network. Therefore, the study of the interrelation between the neural and the symbolic side is of central importance for the field.

We could start a philosophical discussion by replacing artificial neural network by brain and symbolic knowledge processing by mind. Then, the study of neural-symbolic integration is just an instance of the mind-body problem. From this rather abstract point of view we can move towards more concrete problems which are addressed in the area of cognitive science:

- How can symbolic knowledge processing be implemented within a neural system?
- How can symbolic knowledge be learned within a neural system?
- How can symbolic knowledge emerge from activities of a neural system?

These are again just three of the research problems discussed in Section 1.4.

1.1.4 The Neural-Symbolic Cycle

Figure 1.1 shows the *Neural-Symbolic Cycle* which depicts the general approach to the integration followed in this thesis. Starting from a symbolic system, which is both readable and writable by humans, we create a neural (also called connectionist) system by embedding the symbolic knowledge. Those neural systems can then be trained using the powerful connectionist training methods, which allow the modification of the rules by generalisation from raw data. If this learned or refined knowledge is later extracted from the neural system, we obtain a readable version of the acquired knowledge.

In this thesis, we look at some aspects of this integration by following the neural-symbolic cycle. We investigate the embedding of propositional logic programs into standard connectionist systems first and discuss the training and the extraction of propositional rules within and from such a system. We also look at the embedding of first-order rules into standard neural networks.

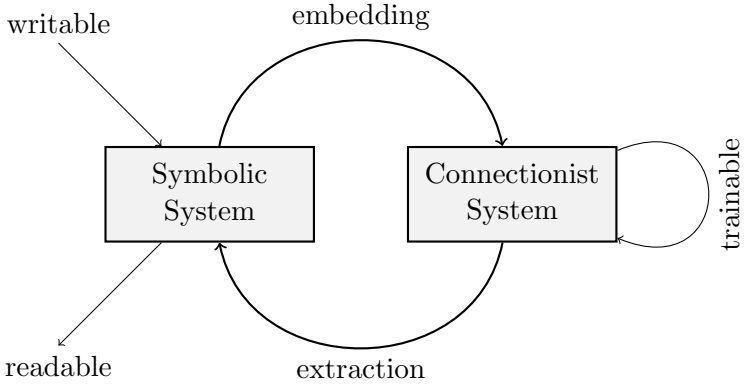


Figure 1.1: *The Neural-Symbolic Cycle*

1.2 Related Work

In this section, we discuss some related work. This includes in particular the results by Warren Sturgis McCulloch and Walter Pitts on the relation between finite automata and connectionist systems, the Core Method, as first proposed by Steffen Hölldobler and Yvonne Kalinke, the KBANN-approach by Geoffrey G. Towell and Jude W. Shavlik and work based on Jordan B. Pollacks recursive auto-associative memories. Further references are given in Section 1.3, where we discuss a new classification scheme for neural-symbolic systems.

1.2.1 Connectionist Systems and Finite Automata

The advent of automata theory and of artificial neural networks, marked also the advent of neural-symbolic integration. In their seminal paper [MP43] Warren Sturgis McCulloch and Walter Pitts have shown that there is a strong relation between symbolic systems and artificial neural networks. In particular, they have shown that for each finite state machine there is a network constructed from binary threshold units – and vice versa – such that the input-output behaviour of both systems coincide. This is due to the fact that simple logical connectives such as conjunction, disjunction and negation can easily be encoded using binary threshold units, with weights and thresholds set appropriately. The main ideas underlying this transformation are illustrated in Example 1.2.1.

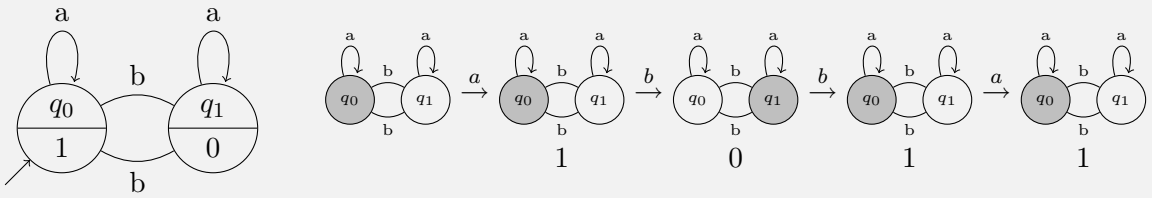
A network of n binary threshold units can be in 2^n different states only, and the change of state depends on the current input to the network only. These states and transitions can easily be encoded as a finite automaton, using a straightforward translation [MP43, Kle56].

An extension to the class of weighted automata is given in [BHS04]. Instead of propagating truth values only, the elements of an arbitrary semiring can be propagated through the network. Using the boolean semiring, we obtain a usual McCulloch-Pitts network, but other instantiations are possible as well, like for example the real numbers (yielding usual $\Sigma - \Pi$-networks) or the stochastic semiring.

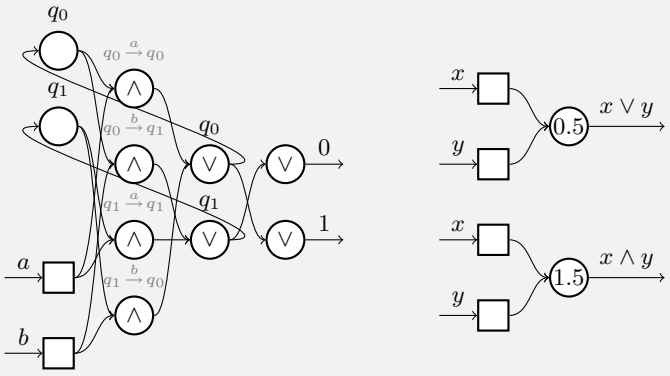
1.2.2 The Core Method

The first work on the approach nowadays called *the Core Method* has been published by Steffen Hölldobler and Yvonne Kalinke in 1994. Their paper *Towards a Massively Parallel Computational Model for Logic Programming* [HK94], marked the beginning of the line of research we follow in this thesis.

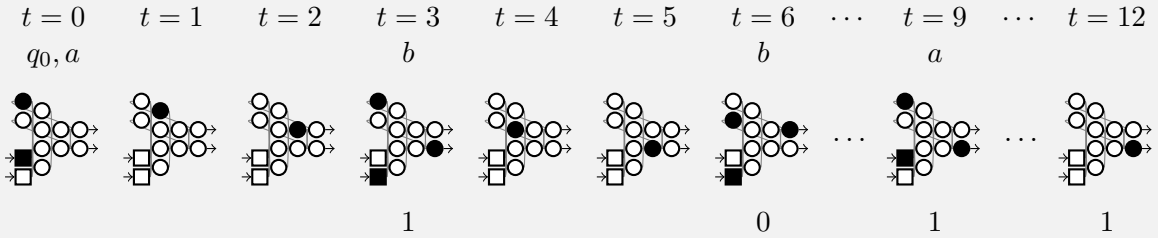
Example 1.2.1 A simple Moore-machine, which is a finite state machine with outputs attached to the states [HU79] is shown below. It consumes inputs a and b, and depending on its internal state q_0 or q_1, it outputs 0 or 1 and changes its state. The processing of the input word *abba* is depicted on the right. The currently active state is marked in gray.



The corresponding McCulloch-Pitts consists of four layers, called *input*, *gate*, *state* and *output*-layer (from left to right in the picture). For each output-symbol $(0, 1)$ there is a unit in the output-layer, and for each input-symbol (a, b) a unit in the lower part of the input-layer. Furthermore, for each state (q_0, q_1) of the automaton, there is a unit in the state-layer and in the upper part of the input layer. In our example, there are two ways to reach the state q_1, namely by being in state q_1 and receiving an 'a', or by being in state q_0 and receiving a 'b'. This is implemented by using a disjunctive neuron in the state-layer receiving inputs from two conjunctive units in the gate layer, which are connected to the corresponding conditions (e.g., being in state q_0 and reading a 'b').



The connectionist implementations of disjunction and conjunction are depicted on the right. All connections have weight 1.0 and the truth-values *true* and *false* are represented as 1.0 and 0.0, respectively. Setting for example $x = \mathit{true} = 1.0$ and $y = \mathit{false} = 0.0$, yields an input of 1.0, which turns $x \vee y$ to be active ($1.0 > 0.5$) and $x \wedge y$ inactive.



A run of the network from while processing the input *abba*. Active units are depicted in black, inactive ones in white. The top-most row shows the time, the row below shows the externally activated units, i.e., at $t = 0$, q_0 and a are activated to initialize the network and provide the first input symbol, respectively. These activations are propagated through the network and yield an output of 1 at $t = 3$ (shown at the bottom). Furthermore, the unit q_0 is activated again, corresponding to the state transition $q_0 \xrightarrow{a} q_0$ of the underlying automaton.

The idea behind the *Core Method* is the usage of feed-forward networks, called *core*, to compute or approximate the meaning function of a logic program. If we then feed the output of the core back as input, we can iterate this function. This meaning function, and in particular its fixed points, capture the semantics of the underlying program. The feed-back can be implemented by recurrent connections between output and input units of the network. This general idea is depicted in Figure 1.2.

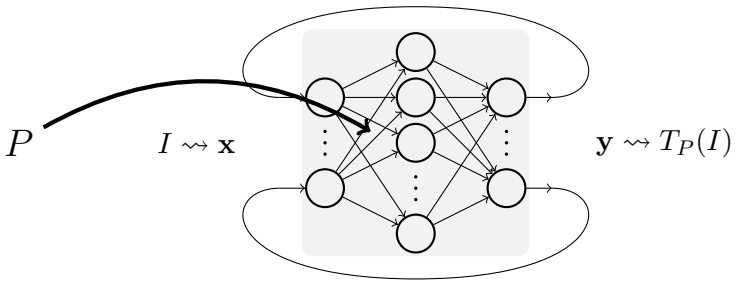


Figure 1.2: *The idea behind the Core Method: A feed-forward network, called core, is used to compute or approximate an embedded of the meaning function T_P for a given logic program P. The recurrent connections between output and input layer allow for an iteration of its application.*

Throughout this thesis, we study a number of algorithms to initialise the core such that it computes a given semantic operators. If the core is also trainable using standard learning algorithms from neural networks, we obtain a system which can be initialised using symbolic knowledge and which can learn, and refine this knowledge, from raw data.

The Early Days of the Propositional Core Method

In [HK94], the authors showed how to construct a feed-forward network of binary threshold units such that the input-output mapping of the network mimics the T_P-operator of a given propositional logic program. We review their approach in Chapter 3.

This work was extended to non-threshold units and other logics. The *Connectionist Inductive Learning and Logic Programming System (CILP)* was first presented by Artur S. d'Avila Garcez, Gerson Zaverucha and Luis Alfredo V. de Carvalho in [dGZdC97, dGZ99]. They showed how to solve one of the main problems preventing the application of the system presented in [HK94]. Inspired by the KBANN-system described below, they used (bipolar) sigmoidal instead of binary threshold units. They gave conditions on the weight matrix of the constructed networks such that the input output mapping can be interpreted as semantic operator of the underlying program. A similar approach is presented in Chapter 3.

Non-Classical Extensions of the Propositional Core Method

Meta-level Priorities for rules have been introduced by Artur S. d'Avila Garcez, Krysia Broda and Dov M. Gabbay in [dGBG00]. By allowing negated conclusions and priorities of rules the authors modelled default reasoning within connectionist networks. In [dGLG02], Artur S. d'Avila Garcez, Luis C. Lamb and Dov M. Gabbay showed the extension of the Core Method to modal logics and in [dGLG03], they presented the Core Method to intuitionistic logics. The treatment of multiple non-classical extensions can be found in [dGGL05a]. A first treatment of multi-valued logics has been presented in [Kal94].

Existential Results for the First-Order Core Method

Steffen Hölldobler, Yvonne Kalinke and Hans-Peter Störr started the research on the first-order core method in [HKS99]. The authors showed that an approximation of the associated meaning function is feasible for certain logic programs. Unfortunately, their proof is non-constructive. But it showed the general applicability of the method to non-propositional programs. The feasibility-result is repeated in Section 7.1 and constructive proofs are presented thereafter.

Pascal Hitzler, Steffen Hölldobler and Anthony Karel Seda review the relation between first-order logic programs and neural networks in [HHS04]. In particular, they extend the class of programs covered in [HKS99] to the class of *strongly determined programs*. Similar extensions are also discussed in [HS00], [Sed05] and [LS06].

Constructive Results for the First-Order Core Method

A first construction of networks to approximate the semantic operator for certain logic programs has been presented in [Wit05, BHW05]. The constructions work for the 1-dimensional embedding as discussed in Section 7.2. All results are presented and extended in Section 7.4.2 and 7.4.3. Those results have first been presented in [BHHW07] and with more details in [BHH08].

Anthony Karel Seda and Máire Lane extend the results presented in [HK94] by allowing the constructions for more general logics [SL05]. Not only 2-valued logics can be used but also logics with finitely many truth values. The authors do also discuss an extension to the first-order case by propositionalising the given first-order program and then using the propositional constructions.

A completely different approach has been taken in [BH04], where we showed how to use results anticipated in [Bad03] to construct radial basis function networks approximating the semantic operator associated to an acyclic logic program. It is based on the close relationship between this operator and a suitable iterated function system (IFS). We could show that the graph of the operator coincides with the attractor of the IFS. Using this relation, we set up a connectionist system implementing the contraction mappings of the IFS. This way, we can use the network to approximate the attractor and, hence, the semantic operator.

Another approach, based on *fibring neural networks* as introduced in [dGG04], has been presented in [BHdG05]. We showed how to represent first-order rules within such fibring networks. Fibring offers modularity, i.e., sub-networks can be trained to solve certain subtasks independently and can be plugged together afterwards. Vladimir Komendantsky and Anthony Karel Seda showed how to compute the set of stable models associated to a normal logic program within a fibring neural network [KS05]. Based on the approach presented in [BHdG05], they construct a family of fibring networks which compute the set of stable models and used a connectionist control structure to iterate through those models.

1.2.3 Knowledge Based Artificial Neural Networks

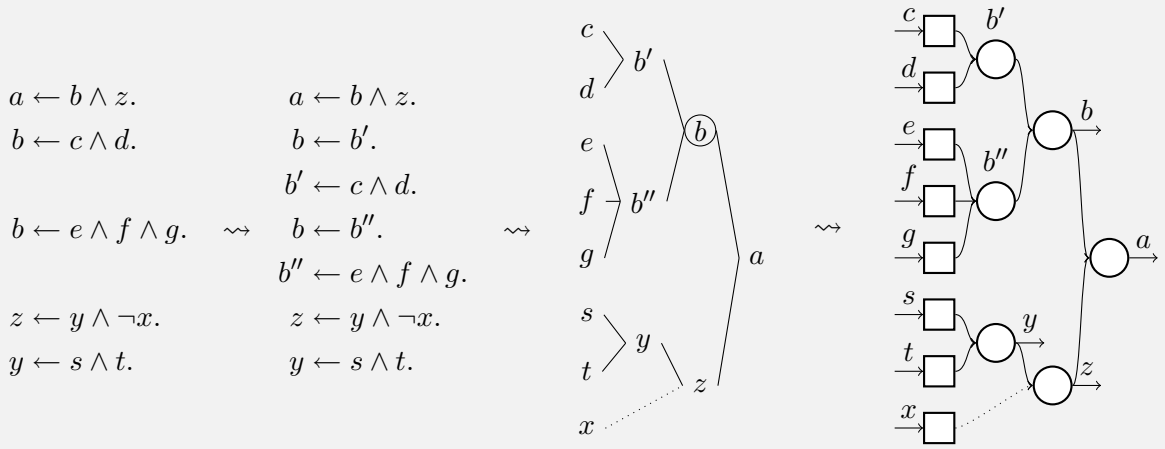
An approach closely related to the Core Method is known as *knowledge based artificial neural networks (KBANN)* introduced by Geoffrey G. Towell and Jude W. Shavlik in [TS94]. They show how to embed a set of logic rules into a connectionist system such that the contained knowledge is preserved.

The authors focus on the refinement of the embedded knowledge using standard training algorithms. KBANN has been evaluated on a number of benchmark problems which show that it outperforms purely symbolic and purely connectionist approaches in those domains.

A set of propositional rules is transformed into an equivalent set of rules, which can easily be embedded into a feed-forward connectionist system. The rules have to be acyclic and are

rewritten into a tree-like structure. This tree can be transformed into a network. The resulting network contains exactly one unit for every propositional variable. Some serving as inputs and others as output units. The weights of the network can be set up such that its behaviour mimics the propagation of truth along the rules. A simple set of rules and the corresponding network are presented in Example 1.2.2.

Example 1.2.2 The following example is taken from [TS94]. The initial set of rules is first transformed into an equivalent one, such that the preconditions of multiple definitions for the same head consist of a single atom. The resulting rules can be represented as a tree, due to the fact that the rules are acyclic. The circled node b represents a disjunction and the dotted line symbolises a negative connection. Finally, the tree is transformed into a neural network. Weights and thresholds of the units are set up such that the units compute either disjunctions or conjunctions.



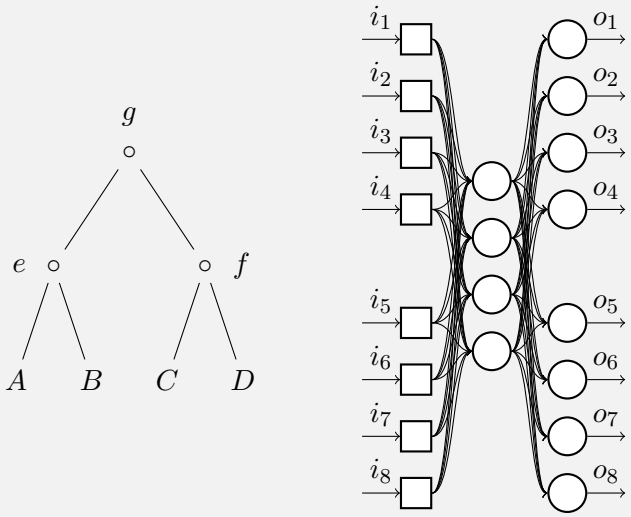
In a final step of the KBANN algorithm, some free hidden units are added and all weights and thresholds are slightly disturbed.

1.2.4 Term Representation using Recursive Auto-Associative Memories

The representation of possibly infinite structures in a finite network is one of the major obstacles on the way to neural-symbolic integration. One attempt to solve this is discussed next, namely the idea of *recursive auto-associative memories (RAAMs)* as introduced in [Pol88, Pol90]. A fixed length representation of variable sized data is obtained by training an artificial neural network using backpropagation. Again, the ideas shall be illustrated by discussing the simple Example 1.2.3.

While recreating the tree from its compressed representation, it is necessary to distinguish terminal and non-terminal vectors, i.e., those which represent leafs of the trees from those representing nodes. Due to noise or inaccuracy, it can be very hard to recognise the "1-of-n"-vectors representing terminal symbols. In order to circumvent this problem different solutions were proposed, which can be found in [SW92, Spe94a, Spe94b]. The ideas described above for binary vectors apply also for trees with larger, but fixed, branching factors, by simply using bigger input and output layers. In order to store sequences of data, a version called S-RAAM (for sequential RAAM) can be used [Pol90]. In [Bla97] modifications were proposed to allow the storage of deeper and more complex data structures than before, but their applicability remains to be shown [Kal97]. Other recent approaches for enhancement have been studied

Example 1.2.3 The following example has been adapted from [Pol90]. A small binary tree which shall be encoded in a real vector of fixed-length is shown below on the left. The resulting RAAM-network is depicted on the right. The network is trained as an encoder-decoder network, i.e. to reproduce the input activations in the output layer [Bis95]. In order to do this, it must create a compressed representation in the hidden layer.



The following table shows the training sample as activations of the layers to be used for training the network. Please note that the training involves a moving target problem, because the internal representation e_i and f_i changes during the training.

input		**hidden**	**output**		**// comments**
1 0 0 0	0 1 0 0	$e_1\ e_2\ e_3\ e_4$	1 0 0 0	0 1 0 0	// $(A,B) \overset{e}{\rightsquigarrow} (A,B)$
0 0 1 0	0 0 0 1	$f_1\ f_2\ f_3\ f_4$	0 0 1 0	0 0 0 1	// $(C,D) \overset{f}{\rightsquigarrow} (C,D)$
$e_1\ e_2\ e_3\ e_4$	$f_1\ f_2\ f_3\ f_4$	$g_1\ g_2\ g_3\ g_4$	$e_1\ e_2\ e_3\ e_4$	$f_1\ f_2\ f_3\ f_4$	// $(e,f) \overset{g}{\rightsquigarrow} (e,f)$

To encode the terminal symbols A, B, C and D we use the vectors $(1,0,0,0)$, $(0,1,0,0)$, $(0,0,1,0)$ and $(0,0,0,1)$ respectively. The representations of e, f and g are obtained during training. After training the network, it is sufficient to keep the internal representation g, because it contains all necessary information for recreating the full tree. This is done by plugging it into the hidden layer and recursively using the output activations, until binary vectors, hence terminal symbols, are reached.

for example in [SSG95, KK95, SSG97, Ham98, AD99], which also include some applications. A survey which includes RAAM architectures and addresses structured processing can be found in [FGKS01]. The related approach on *Holographic reduced representations (HRRs)* [Pla91, Pla95] also uses fixed-length representations of variable-sized data, but using different methods.

1.2.5 More Propositional Approaches

There exist a number of approaches closely related to the Core Method and the KBANN-system in that they are using some kind of activity spreading. Single units represent the truth value of a given proposition and the connections are set up such that the network implements a semantic operator.

Robert Andrews and Shlomo Geva discuss in [AG99] their *Rapid Backprop Networks (RBP)* and show that their initialisation has a positive effect on learning speed, accuracy and generates smaller networks.

In [BG95], Enrico Blanzieri and Attilio Giordana show how propositional rules can be embedded into locally receptive field networks (LRFN). Those networks are a mixture of RBF networks and fuzzy networks. The authors showed how to embed rules into such networks and how to train it. Unfortunately, they did not compare their initialised networks with other approaches.

A different approach has been taken by Gadi Pinkas. In [Pin91b], he shows how to construct a symmetric network (also known as Hopfield network) for a given set of rules. The stable states of those networks, which can be computed by an energy minimisation algorithm, correspond to proofs for a given query.

1.2.6 More Relational Approaches

Relational approaches include those system for which the symbolic side contains first-order predicates and variables but only a finite number of constants and no other function symbols. I.e., the underlying language is still finite, in the sense that only finitely many ground instances can be built.

Humans can solve a wide variety of task very quickly. This type of reasoning has been called *reflexive reasoning* by Shastri and Ajjanagadde. The SHRUTI-system [SA93] provides a connectionist implementation of reflexive reasoning. Relational knowledge is encoded in a network. The binding between variables and constants has been done by a so called phase coding mechanism. The activity is propagated through the network at certain time intervals. If a unit representing a variable and one representing a constant are active at the same time then both are assumed to be bound. Further enhancements allow the usage of negation and inconsistent rules [Sha99, SW99]. Learning capabilities have been added in [SW03, WS03]. The problem of instantiating a rule multiple times has been addressed in [WS04]. Those general problems in the field are discussed in detail in Section 1.4.

Rodrigo Basilio, Gerson Zaverucha and Valmir C. Barbosa describe in [BZB01] their representation of rules within ARTMAP networks and show that their FOCA-system outperforms other approaches on some ILP problems. They basically implement an ILP system using special neurons, which are capable of manipulating tuples of terms.

In [HKW99], Steffen Hölldobler, Yvonne Kalinke and Jörg Wunderlich present a connectionist system to generate the least model of a given datalogic program. The presented system, called BUR calculus, is a "bottom-up" system comparable with the SHRUTI-approach. It is sound and complete for unary predicates but may be unsound otherwise.

1.2.7 More First-Order Approaches

An approach based on the connection method by Wolfgang Bibel [Bib87] has been presented by Steffen Hölldobler and Franz Kurfess in [HK92]. In their CHCL-system they construct the so-called spanning set for a given first order formula. This spanning set can be though of as a set of conditions. A solution is found if all literals occurring connected in this set can be unified simultaneously. The unification is performed using a connectionist unification algorithm presented in [Höl90].

The approach based symmetric networks pursued by Gadi Pinkas described above, has been extended to first-order logic. In [Pin94], he showed how to compute proofs for first-order queries with respect to a given knowledge base. The networks are constructed for a given set of rules and the maximal length of the constructible proofs.

A completely different idea has been followed by Helmar Gust and Kai-Uwe Kühnberger in [GK05]. Logical formulae are transformed into a variable free representation, which is based on so-called Topoi, a certain type of categories. The author exploit the fact that logic formulae as well as arrows in category theory possess some form of compositionality. This results in a commuting diagram with respect to the composition of arrows. A neural network is set up to learn the representation of the arrows and their compositions, such that all equations resulting from the commuting diagram are fulfilled. After training the network it can be queried by comparing the arrow corresponding to some query with the arrows representing truth and falsity. The authors showed that the system is able to learn and to derive valid conclusions from a given initial theory.

1.2.8 Connectionist Learning and Symbolic Knowledge

The positive effect of initialising connectionist systems with available background knowledge is known since 1994 when Geoffrey G. Towell and Jude W. Shavlik described their first results of the KBANN-system in [TS94]. In fact their system has been designed to study this effect. And they showed that it outperforms purely symbolic and purely connectionist classifiers. A similar effect was shown by Robert Andrews and Shlomo Geva in [AG99] for their rapid backpropagation networks. Artur S. d'Avila Garcez and colleagues showed the same positive effects using their CILP-system mentioned above.

The rules embedded into the connectionist systems are usually hand-crafted by some human expert. A different approach has been taken in [dCNFR04]. The authors study the effect of using rules acquired by other machine learning algorithms, like ID3. They could show that those rules can be used as well as hand-crafted rules and that the connectionist system could further improve those rules. I.e., even though the rule-generating approach converged to some solution, this solution has been further improved using the connectionist learning.

In [JBN04], R. P. Jagadeesh, Chandra Bose and G. Nagaraja compare the performance of the original KBANN training, i.e., normal backpropagation with more advanced algorithms. In particular, they compared it to R-Prop and found that error and training time can be reduced.

1.2.9 Rule Extraction from Connectionist Systems

The extraction of rules from trained networks is the problem of finding symbolic rules describing the behaviour of the network. Nice introductions to the area and to existing techniques can be found in [ADT95] and [Jac05].

Rule-extraction from multi-layer perceptrons has for example been described in [DAG98, DAG01] The extraction of rules from radial basis function networks has for example been investigated in [MTWM99].

The new pedagogical approach presented in Section 6.2 is in parts related to the Combo algorithm as presented by Ramanathan Krishnan, G. Sivakumar and Pushpak Bhattacharya in [KSB99]. The authors use a search tree, which is ordered wrt. the minimal input on each level. I.e., they need to construct all tree nodes for a given level and then to sort them accordingly. In the approach presented below, we do neither need to construct all those nodes, nor to sort them.

In [dGBG01], Artur S. d'Avila Garcez, Krysia Broda and Dov M. Gabbay present a method for the extraction of propositional logic rules from neural networks. It is based on a specific ordering on the space of interpretations which allows for an efficient extraction of the learnt rules. The rule language does also allow so-called m-of-n rules.

1.3 A Classification Scheme for Neural-Symbolic Systems

In [BH05], we suggested a classification scheme for neural-symbolic systems. Here, we review this scheme and discuss some systems. By introducing this classification scheme, we intended to bring some order to the heterogeneous field of research, whose individual approaches are often largely incomparable. We suggested to use a scheme consisting of three main axes, called *Interrelation*, *Language* and *Usage*. We roughly followed the scheme introduced and discussed in [Hil95, HP04] where the authors try to depict each system at exactly one point in a taxonomic tree. But certain properties or design decisions of systems are rather independent, and should be understood as different *dimensions*. The resulting three main dimensions are shown in Figure 1.3.

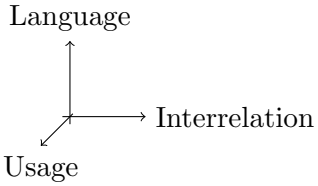


Figure 1.3: *The three main dimensions of our classification scheme*

The interrelation-dimension, depicted in Figure 1.4, divides the approaches into two main classes, namely into *integrated* (also called *unified* or *translational* in [Hil95, HP04]) and *hybrid* systems. Integrated are those, where full symbolic processing capabilities emerge from neural structures and processes. They can be further subdivided into *neural* and *connectionist* approaches. Neural indicates the usage of neurons which are very closely related to biological neurons. In connectionist approaches there is no claim to biological plausibility, instead general artificial neural network architectures are used. Depending on their architecture, they can be split into *standard* and *non-standard* networks. Furthermore, we can distinguish *local* and *distributed* representation of the knowledge. The interrelation dimension is discussed in detail in Section 1.3.1.

Note that the subdivisions of the axis are again independent of each other and they should be understood as independent sub-dimensions. We understand the neural-connectionist dimension as a subdivision of integrated systems, and the distributed-local and standard-non-standard dimensions as independent sub-divisions of connectionist systems. Arrows within the diagrams indicate a detailed discussion of the connected issues below.

Figure 1.5 depicts the second axis in our scheme. Here, the systems are divided according to the language used in their symbolic part. This includes automata, grammars and (nested)

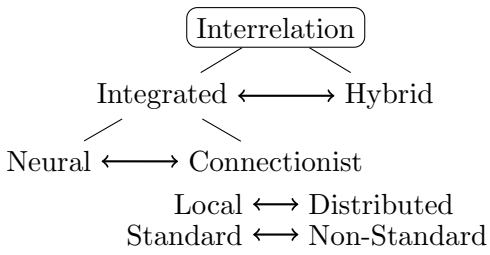


Figure 1.4: *The details of the interrelation dimension: Systems can be divided into hybrid and integrated approaches, which can be classified as neural or connectionist, depending on their biological plausibility. The latter can be further partitioned into local and distributed approaches, depending on the internal representation, and as standard or non-standard with respect to the type of units.*

terms as well as logic formulae, which are further divided into propositional, relational and first-order languages. The language axis is discussed in more detail in Section 1.3.2.

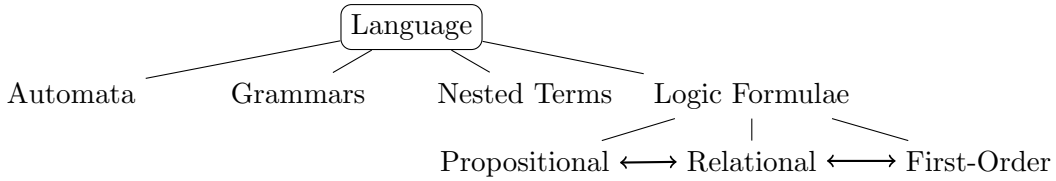


Figure 1.5: *The details of the language dimension: Different languages used in the symbolic part of neural-symbolic systems.*

Most systems focus on one or only a few aspects of the neural-symbolic learning cycle depicted in Figure 1.1, i.e., either the representation of symbolic knowledge within a connectionist setting, or the training of initialised networks, or the extraction of symbolic systems from a network. Depending on this main focus we can distinguish the systems as shown in Figure 1.6. The issues of *extraction* versus *representation* on the one hand and *learning* versus *reasoning* on the other hand, are discussed in Section 1.3.3.

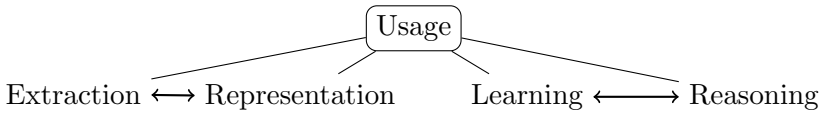


Figure 1.6: *The details of the usage dimension: Different focus within neural-symbolic systems.*

1.3.1 Interrelation between Neural and Symbolic Part

The interrelation between the symbolic and the connectionist side is discussed by describing the identified contrasts from Figure 1.4, i.e., by describing the connection between both parts and the type of connectionist system under consideration.

Integrated versus Hybrid

Hybrid systems are characterised by the fact that they combine two or more problem-solving techniques in order to address a problem, which run in parallel. An *integrated* neural-symbolic

system differs from a hybrid one in that it consists of one connectionist main component in which symbolic knowledge is processed. Integrated systems are sometimes also referred to as *embedded* or *monolithic* hybrid systems [Sun01].

Neural versus Connectionist

Two driving forces behind the field of neural-symbolic integration are on the one hand an understanding of human cognition, and on the other the vision of combining connectionist and symbolic artificial intelligence technology in order to arrive at more powerful reasoning and learning systems for computer science applications.

In [MP43] the motivation for the study was to understand human cognition, i.e., to pursue the question how higher cognitive – logical – processes can be performed by artificial neural networks. In this line of research, the question of *biological feasibility* of a network architecture is prominent, and inspiration is often taken from biological counterparts.

The SHRUTI system [SA93], for example, addresses the question how it is possible that biological networks perform certain reasoning tasks very quickly. Indeed, for some complex recognition tasks which involve reasoning capabilities, human responses occur sometimes at *reflexive* speed, particularly within a time span which allows processing through very few neuron layers only.

The spiking neurons networks [Maa02] take an even more realistic approach to the modelling of temporal aspects of neural activity. Neurons, in this context, are considered to fire so-called *spike trains*, which consist of patterns of firing impulses over certain time intervals. The complex propagation patterns within a network are usually analysed by statistical methods. The encoding of symbolic knowledge using such temporal aspects has hardly been studied so far, an exception being [Sou01]. To date, only a few preliminary results on computational aspects of spiking neurons have been obtained [NM02, MM04, MNM05]. An attempt to learn logical rules within a network of spiking neurons is described in [Dar00].

[vdVdK05], shows how natural language could be encoded using biologically plausible models of neural networks. The results appear to be suitable for the study of neural-symbolic integration, but it remains to be investigated to which extent the provided approach can be transferred to symbolic reasoning.

The lines of research just reviewed take their major motivation from the goal to achieve biologically plausible behaviour or architectures. As already mentioned, neural-symbolic integration can also be pursued from a more technically motivated perspective, driven by the goal to combine the advantages of symbolic and connectionist approaches by studying their interrelationships. The work on the Core Method and KBANN, as discussed above, can be subsumed under this technologically inspired perspective. Neither all what follows in later chapters is not meant to be biologically plausible.

Local versus Distributed Representation of Knowledge

For integrated neural-symbolic systems, the question is crucial how symbolic knowledge is represented within the connectionist system. If standard networks are being trained using backpropagation, the knowledge acquired during the learning process is spread over the network in diffuse ways, i.e., it is in general not easy or even possible to identify one or a small number of nodes whose activations contain and process a certain symbolic piece of knowledge.

The RAAM architecture [Pol88] and their variants are clearly based on distributed representations. Technically, this stems from the fact that the representation is initially learned, and no explicit algorithm for translating symbolic knowledge into the connectionist setting is being used.

Most other approaches to neural-symbolic integration, however, represent data locally. E.g., the SHRUTI-System associates a defined node assembly to each logical predicate, and the architecture does not allow for distributed representation. The approaches for propositional connectionist model generation using the Core Method encode propositional variables as single nodes in the input and output layer, and logical formulae (rules) by single nodes in the hidden layer of the network.

The design of a distributed encoding of symbolic data appears to be particularly challenging. It also appears to be one of the major bottlenecks in producing applicable integrated neural-symbolic systems with learning and reasoning abilities. This becomes apparent in the difficulties faced while extending the Core Method to first-order logic. Symbolic entities are not represented directly. Instead, interpretations (i.e., valuations) of the logic are represented. Concrete representations, as discussed in Chapter 7, distribute the encoding of the interpretations over several nodes, but in a diffuse way. This encoding results in a distributed representation. Similar considerations apply to the recent proposal [GK05], where first-order logic is first converted into variable-free form (using topoi from category theory), and then fed to a neural network for training.

Standard versus Non-standard Network Architecture

Even though neural networks are a widely accepted paradigm in AI it is hard to make out a standard architecture. But, all so-called standard-architecture systems agree at least on the following:

- real numbers are propagated along the connections
- units compute very simple functions only
- all units behave similarly (i.e., they use similar simple functions and the activation values are always within a small range)
- only simple recursive structures are used (e.g., connecting only the output back to the input layer, or use self-recursive units only)

When adhering to these standard design principles, powerful learning techniques as for example backpropagation [Wer74, RHW86] or Hebbian Learning [Heb49] can be used to train the networks, which makes them applicable to real world problems.

However, these standard architectures do not easily lend themselves to neural-symbolic integration. In general, it is easier to use non-standard architectures in order to represent and work with structured knowledge, with the drawback that powerful learning abilities are often lost.

Neural-symbolic approaches using standard networks are for example the CILP system [dGZ99], KBANN [TS94], RAAM [Pol88] and [SL05]. Usually, they consist of a layered network, consisting of three or in case of KBANN more layers, and sigmoidal units are being used. For these systems experimental results are available showing their learning capabilities. As discussed above, these systems are able to handle propositional knowledge (or first-order with a finite domain).

Non-standard networks were used e.g., in the SHRUTI system [SA93], the approaches bases on fractal geometry [BH04] and those based on fibring networks [BHdG05]. In all these implementations non-standard units and non-standard architectures were used, and hence none of the usual learning techniques are applicable. However, for the SHRUTI system limited learning techniques based on Hebbian Learning [Heb49] were developed in [Sha02, SW03, WS03].

1.3.2 Language Underlying the Symbolic Part

Symbolic approaches include the relation to automata as in [MP43], to grammars [Elm90, Fle01, FCBGM99] or to the storage and retrieval of terms [Pol90], whereas the logical approaches require either propositional or first-order logic systems, as e.g. in [HK94] or [BHH08].

One of the motivations for studying neural-symbolic integration is to combine connectionist learning capabilities with symbolic knowledge processing. While the main interest in this thesis are logical aspects of symbolic knowledge, this is not necessarily always the main focus of investigations.

Work on representing automata or weighted automata [Kle56, MP43, BHS04] using artificial neural networks, for example focuses on computationally relevant structures, such as automata, and not directly on logically encoded knowledge. Nevertheless, such investigations show how to deal with structural knowledge within a connectionist setting, and can serve as inspiration for corresponding research on logical knowledge.

Recursive auto-associative memory, RAAM, and their variants deal with terms only, and not directly with logical content. A RAAM allows connectionist encodings of first-order terms, where the underlying idea is to present terms or term trees sequentially to a connectionist system which is trained to produce a compressed encoding characterised by the activation pattern of a small collection of nodes.

A considerable body of work exists on the connectionist processing and learning of structured data using recurrent networks [SSG95, SSG97, FGKS01, Ham02, Ham03, HMSS04a, HMSS04b]. The focus is on tree representations and manipulation of the data. A treatment of graphs within so-called *GraphSOM (Graph Self-Organising Maps)* has been presented in [HTSK08]. The authors show how to cluster a large dataset of XML-documents using their GraphSOM model. In [SGT$^+$08], the authors propose the *graph neural network*, a type of artificial neural network that is able to process data represented as graphs directly. Every node of a graph is represented as a simple feed-forward network. Hence, a graph is embedded into a recurrent network. The authors also show how to train the network and discuss some applications indicating a very good performance of the approach.

[HKL97, KL98] study the representation of counters using recurrent networks, and connectionist unification algorithms as studied in [Höl90, HK92, Höl93] are designed for manipulating terms, but already in a clearly logical context. The representation of grammars [GCM$^+$91] or more generally of natural language constructs [vdVdK05] also has a clearly symbolic (as opposed to logical) focus.

It remains to be seen, however, to what extent the work on connectionist processing of structured data can be reused in logical contexts for creating integrated neural-symbolic systems with reasoning capabilities. Integrated reasoning systems like the SHRUTI-system (Section 1.2.6) and those presented in Chapter 7 lack the capabilities of the term-based systems, so that a merging of these efforts appears to be a promising albeit challenging goal.

Propositional versus Relational versus First-Order

Logic-based integrated neural-symbolic systems differ with respect to the knowledge representation language they are able to represent. Concerning the capabilities of the systems, a major distinction needs to be made between those which deal with propositional logics, and those based on first-order predicate logics. Relational languages (also referred to as Datalog) are those which contain first-order variables but no function symbols and only finitely many constants.

What we mean by propositional logics in this context includes propositional modal, temporal, non-monotonic, and other non-classical logics. One of their characteristic feature which

distinguishes them from first-order logics for neural-symbolic integration is the fact that they are of a finitary nature: propositional theories in practice involve only a finite number of propositional variables, and corresponding models are also finite. Also, sophisticated symbol processing as needed for nested terms in the form of substitutions or unification is not required.

Due to their finiteness it is thus fairly easy to implement propositional logic programs using neural networks [HK94] (Chapter 3). A considerable body of work deals with the extension of this approach to non-classical logics [dGBG00, dGLG02, dGLG03, dGGL04, dGLBG04, dGGL05a, dGGL05b]. Earlier work on representing propositional logics is based on Hopfield networks [Pin91b, Pin91a]. An approach for nonmonotonic reasoning based on so-called inhibition nets has been presented by Hannes Leitgeb in [Han01].

In contrast to this, predicate logics allow to use function symbols as language primitives. Consequently, it is possible to use terms of arbitrary depth, and models necessarily assign truth values to an infinite number of ground atoms. The difficulty in dealing with this in a connectionist setting lies in the finiteness of neural networks, necessitating to capture the infinitary aspects of predicate logics by finite means. The first-order approaches presented in [HKS99, HS00, BH04, HHS04, BHdG05, BHW05] solve this problem by using encodings of infinite sets by real numbers, and representing them in an approximate manner. They can also be carried over to non-monotonic logics [Hit04].

[BHdG05], which builds on [dGG04] and [Gab99] uses an alternative mechanism in which matching of terms is controlled via fibring. More precisely, certain network constructs encode the matching of terms and act as gates to the firing of neurons whenever corresponding symbolic matching is achieved.

A prominent sub-problem in first-order neural-symbolic integration is that of variable binding. It refers to the fact that the same variable may occur in several places in a formula, or that during a reasoning process variables may be bound to instantiate certain terms. In a connectionist setting, different parts of formulae and different individuals or terms are usually represented independently of each other within the system. The neural network paradigm, however, forces sub-nets to be blind with respect to detailed activation patterns in other sub-nets, and thus does not lend itself easily to the processing of variable bindings.

Research on first-order neural-symbolic integration has led to different means of dealing with the variable binding problem. One of them is to use temporal synchrony to achieve the binding. This is encoded in the SHRUTI system, where the synchronous firing of variable nodes with constant nodes encodes a corresponding binding. Other approaches, as discussed in [BS99], encode binding by relating the propagated activations, i.e. real numbers.

Other systems avoid the variable binding problem by converting predicate logical formulae into variable-free representations. The approaches in [HKS99, HS00, HHS04, Hit04, Sed05, SL05, BHdG05, BHW05] make conversions to (infinite) propositional theories, which are then approximated. [GK05] use topos theory instead. The approach presented in [Bor96] could lead to a novel encoding of first-order interpretations, in which the authors construct a finite representation for (usually) infinite interpretations.

It shall be noted here that SHRUTI addresses the variable binding problem, but allows to encode only a very limited fragment of first-order predicate logic [HKW99]. In particular, it does not allow to deal with function symbols, and thus could still be understood as a finitary fragment of predicate logic.

1.3.3 Usage of the System

As mentioned above, most systems address only certain parts of the neural-symbolic cycle. They are usually concerned with either representation of extraction and with either learning or reasoning. Those contrasts are discussed next.

Extraction versus Representation

It is apparent, that both the representation and the extraction of knowledge are of importance for integrated neural-symbolic systems. They are needed for closing the neural-symbolic learning cycle as shown in Figure 1.1. However, they are also of independent interest, and are often studied separately.

As for the representation of knowledge, this component is present in all systems presented so far. The choice how representation is done often determines whether standard architectures are used, if a local or distributed approach is taken, and whether standard learning algorithms can be employed.

A large body of work exists on extracting knowledge from trained networks, usually focusing on the extraction of rules. [Jac05] gives a recent overview over extraction methods. In [GCM+91], a method is given to extract a grammar represented as a finite state machine from a trained recurrent neural network. [MTWM99] show how to extract rules from radial basis function networks by identifying minimal and maximal activation values. Some of the other efforts are reported in [TS93, ADT95, Bol00, dGBG01, LBH05]

It shall be noted that only a few systems have been proposed to date which include representation, learning, and extraction capabilities in a meaningful way, one of them being CILP [dGZdC97, dGZ99, dGBG01]. It is still an open and difficult research challenge to provide similar functionalities in a first-order setting.

Learning versus Reasoning

Ultimately, one goal is to create an effective AI system with added reasoning and learning capabilities, as recently pointed out by Valiant [Val03] as a key challenge for computer science. It turns out that most current systems have either learning capabilities or reasoning capabilities, but rarely both. SHRUTI, for example, is a reasoning system with very limited learning support.

In order to advance the state of the art in the sense of Valiant's vision, it is necessary to create systems with combined capabilities. In particular, learning should not be independent of reasoning, i.e., initial knowledge and logical consequences thereof should help guiding the learning process. Chapter 5 contains a first attempt in this direction.

1.4 Challenge Problems

In [Höl00, BHH06] nine challenge problems have been identified, which need to be solved to create a fully integrated system for first-order reasoning within a connectionist system. Because most of them are general problems within the field of neural-symbolic integration, we re-discuss them here to further motivate the study of this subject.

1.4.1 How Can (First-Order) Terms be Represented in a Connectionist System?

The question of a connectionist representation of terms is of course the primary one which needs to be answered, not only for first-order terms but for knowledge represented in any symbolic language.

In propositional settings, it is common to associate a designated unit to every propositional variable. This was done, for example, in all propositional Core Method approaches (Section 1.2.2 & Chapter 3), in the KBANN-system (Section 1.2.3), and in symmetric networks representing propositional formulae [Pin91b]. This association is possible because there are usually only finitely many propositional variables involved in a problem, and hence every one

of them can be associated with a unit in the input and output layer. By activating those units in the input layer for which the propositions are considered to be true, we can feed propositional interpretations into the network. Analogously, we can understand the output units of the network as the resulting interpretations, which follows from the input with respect to the function implemented in the network. It turns out to be rather straight-forward to embed propositional rules into such networks by initialising the internal units such that they implement simple logical connectives like conjunction and disjunction. We discuss a propositional setting in detail in Chapter 3.

Unfortunately, the situation is far more complex when going to first-order logic, because the underlying language contains usually infinitely many atomic (ground) propositions. Therefore, we would need infinitely many units while using the above mentioned direct association between units and propositions.

In the SHRUTI and BUR systems mentioned above, a phase coding mechanism was used to encode first-order formulae. I.e., to each element of the underlying universe a time-slot is associated. Activating a unit corresponding to an argument of a given predicate within the time-slot for a given element means that the argument should be instantiated to that element. Therefore, the universe must contain only finitely many elements, otherwise we would need infinitely many time steps.

Vectors of fixed length have been used for the auto-associative memories as described in Section 1.2.4. Instead of fixing a mapping between inputs and activations, those vectorial representations of terms have been acquired during a training phase.

In hybrid systems (see Section 1.3.1), the terms are represented and manipulated in a conventional symbolic way. As mentioned above, this is not the kind of integration we are hoping for. Nonetheless, the problem of the embedding remains as it must be solved to design the interface between the symbolic and the connectionist subsystem.

In Chapter 7, we discuss an new approach in which first-order interpretations are mapped to (vectors of) real numbers which can then be propagated into the networks. If the networks are set up appropriately, we can read of approximations of the consequences as an encoded first-order interpretation from the output units.

None of the above mentioned approaches is completely satisfactory yet. Therefore, new ideas for the embedding of symbolic terms are still necessary, in particular for the first-order setting.

1.4.2 Can (First-Order) Rules be Extracted from a Connectionist System?

The extraction of rules is somewhat the opposite problem to the representation. The goal of rule-extraction methods is the creation of (human readable) sets of rules that capture the behaviour of the network. (In Section 6.1 this is discussed more formally.) The extraction of rules is necessary to gain access to the knowledge acquired during the training of the network. And it allows the completion of the neural-symbolic cycle discussed above.

There exist a number of propositional approaches. E.g., [TS93], [KSB99] and [dGBG01]. Another new approach (slightly related to [KSB99]) is presented in full detail in Chapter 6. Unfortunately, the constructed propositional rule sets tend to be very large and are, hence, incomprehensible to humans. Therefore, extraction methods producing richer logics are required.

So far, there are no approaches really beyond propositional logic. All rules which can be extracted from a trained network so far are in principle propositional rules and it is still completely unclear how first-order rules can be extracted. In Chapter 7, we discuss methods to initialise and train networks with first-order knowledge. We know that it is possible to obtain networks approximating the semantic operators of first-order programs. But once the

knowledge is embedded, it is hidden in the weights. Therefore, completely new methods are required to obtain access to the rules embedded in the networks.

1.4.3 Can Neural Knowledge Representation be Understood Symbolically?

While the embedding of rules into connectionist systems does usually lead to a local representation within the system, it is assumed that standard learning procedures lead to a distributed representation. I.e., there is not a single hidden node responsible for a learned rule. The rules are rather distributed over all hidden units. Looking at the activation values of those hidden units in constructed networks, we find that only one of the nodes is active at a time. In networks trained from raw data, we find that the hidden units are all activated but their activation pattern varies for different inputs. How this distributed representation of knowledge can be interpreted in a symbolic manner is also completely unclear yet.

1.4.4 Can Learning Algorithms be Combined with Symbolic Knowledge?

Most neural-symbolic systems follow the Neural-Symbolic cycle presented in Figure 1.1. Available background knowledge is embedded into a connectionist system. This is then trained using raw data and the acquired knowledge is extracted thereafter. It has been shown that the a-priori embedding of rules leads to faster and better convergence during the training. The network and, in particular, its training is treated as a black box. A different approach is not to embed the knowledge prior to the training but use it to influence the training process itself. How this can be done has been unclear until we have presented a first approach in [BHM08]. The results presented there are repeated in Chapter 5.

As before, the situation becomes worse for first-order logic. Most existing systems into which first-order rules can be embedded are non-standard architecture. I.e., non-sigmoidal units and new methods to propagate activations through the network are used. Therefore, the well known training algorithms like backpropagation are not applicable any more. On the one hand we can try to build standard-networks and on the other hand we could try to modify the training algorithms to better suit this symbolic setting. Both approaches are pursued in Chapter 7.

1.4.5 Can Multiple Instances of First-Order Rules be Used in Connectionist Systems?

A problem of first-order reasoning is that it cannot be determined in advance how many instances of a rule are necessary to answer a given query. While embedding rules into connectionist systems, this problem needs to be addressed. There exist different solutions to the problem:

Restrict the number of instances in advance: This has been done, for example, in the CHCL- and SHRUTI-system discussed above. In both the number of copies has been fixed.

Ignore the problem: The BUR-system does not provide multiple copies. This is the reason for its unsoundness if multi-place relations are involved.

Circumvent the problem: In the approach followed in Chapter 7, the creation of multiple instances is mapped to the problem of obtaining better approximations. By constructing all ground clauses necessary to obtain a given accuracy, we obtain a solution to this problem.

Unfortunately, in all presented solutions, we need to fix either the number of copies or the level of approximation in advance. A system which can automatically adjust those numbers depending on the problem would be desirable. In a fully integrated systems, this adaptation should be done on a purely connectionist level, i.e., without intervention from the symbolic part. But how this can be achieved is also completely unclear.

1.4.6 Can Insights from Neuroscience be Used to Design Better Systems?

As mentioned in Section 1.1.2, a goal of the study of neural-symbolic integration is to gain insights into neuroscience and biological plausibility. This includes the following questions:

- Can the accumulation of electric potential within the dendrite of a biological neuron be understood from a logical perspective?
- Can we develop methods to understand the temporal aspects of transmissions between different biological neurons?
- Can we assign a symbolic meaning to firing patterns of collections of neurons?

Whether the study of neural-symbolic integration can help to answer this question, remains open. But because looking at biological systems usually improves artificial systems, we might get new ideas to design better neural-symbolic systems. The spiking neuron approach mentioned above is biologically more plausible and has been inspired by neuroscientific findings. But so far there are no successful neural-symbolic systems based on those networks.

1.4.7 Can the Relation Between Neural-Symbolic and Fractal Systems be Exploited?

In Section 1.2.2, we mentioned the relation between fractal geometry and the embedding of first-order rules into connectionist systems. This is not a singular occurrence of this relation. Howard A. Blair et al. reported on similar results for the dynamics of other symbolic systems [BDJ+99]. Fractal geometry, or topological dynamics might open possibilities to capture the dynamics of the underlying processes and to bridge the gap between the discrete world of symbolic computation and the continuous world of neural networks. Unfortunately, research in this direction is difficult due to the fact that all related subjects are equally hard to study.

1.4.8 What does a Theory for the Neural-Symbolic Integration Look Like?

All approaches achieved in the field of neural-symbolic integration are more or less unrelated to each other. Different symbolic systems are mapped to different connectionist systems. The choice of the approach is made by the researcher and usually completely new ideas are developed for new application domains.

One goal of the study in this field is to establish a unifying theory connecting all existing approaches. This theory should dictate which neural-symbolic approach is to be used for a given problem. More complex problems require more powerful neural-symbolic systems. To create such a theory, we need to fully and formally characterise the properties of the different approaches. This includes:

- Showing the exact relation between the symbolic and the connectionist system.
- Relating the different integrated systems to each other, e.g., by showing that one is able to simulate the other.

There is hope that such a theory allows to gain a deeper understanding of the field.

1.4.9 Can Neural-Symbolic Systems Outperform Conventional Approaches?

From an application perspective, we would like to develop a system, which conjoins the advantages of symbolic and connectionist approaches and outperforms conventional systems based on either of the paradigms. Experimental results support the hope that integrated systems are able to outperform other approaches. Already the first results by Geoffrey G. Towell and Jude W. Shavlik in [TS94] show that their KBANN-system achieves better classification results than purely symbolic and purely connectionist algorithms. Similar results have been reported for the CILP-system by Artur S. d'Avila Garcez, Gerson Zaverucha and Luis Alfredo V. de Carvalho in [dGZdC97]. But the problems addressed in those reports have been relatively small benchmark problems from the UCI Machine Learning Repository [AN07].

A real world example is presented in Chapter 4, where we showed how to solve the word tagging problem using a neural-symbolic system. The data-sets used in this application domain have been magnitudes bigger than previous problems. Nonetheless, it is yet a hand-crafted system, specifically designed to solve one particular problem. Therefore, the good performance might be not too surprising.

The question whether neural-symbolic systems can outperform existing approaches remains to be answered. Of course we hope that this is indeed the case.

1.5 Structure of this Thesis

In this chapter, we have motivated the study of neural-symbolic integration. We have also discussed a number of related approaches and a classification scheme for those systems. This scheme has first been proposed in *"Dimensions of neural-symbolic integration — a structured survey"* [BH05]. Finally, we have discussed some challenge problems which need to be solved, and which in part are solved within this thesis. Those challenge problems have been proposed and discussed first in *"The Integration of Connectionism and First-Order Knowledge Representation and Reasoning as a Challenge for Artificial Intelligence"* [BHH06].

Chapter 2 introduces all basic notions and concepts required for later chapters. This includes some basic concepts from mathematics, in particular metric spaces and contractive mappings, iterated function systems, logic programs and different connectionist architectures.

Chapter 3 to 6 cover the propositional Neural-Symbolic cycle. Starting with the embedding of propositional rules into connectionist systems, the training of those with respect to symbolic background knowledge, and finally in Chapter 6, the extraction of propositional rules from trained networks. We discuss the embedding of propositional rules in Chapter 3. Even though this embedding has been presented before [HK94, dGBG02], we discuss it from a more general perspective. Some restrictions imposed before have been softened. This may lead to a uniform treatment of all propositional approaches presented so far.

In Chapter 4 an application of a neural-symbolic system to the problem of part of speech tagging is presented. We show how to set up a network such that it assigns the most likely grammatical tag (word-class) to a given word with respect to the word's context. Furthermore, we show how to embed grammatical background knowledge into such a network and find that the training performance of the network can be increased.

The modified training scheme presented in Chapter 5 has first been published in *"Guiding Backprop by Inserting Rules"* [BHM08]. The method is based on a repeated insertion of error-correcting rules during the training process. This leads to an improved performance of the network. Even though only a first experiment is presented, the approach itself is not dependent on the particular application domain but can be applied in other settings alike. To the best of our knowledge, it is the first approach which allows the incorporation of symbolic rules into the training process.

The extraction procedure described in Chapter 6, has been discussed in parts in *"Extracting Propositional Rules from Feed-forward Neural Networks — A New Decompositional Approach"* [BHME07] and *"Extracting Propositional Rules from Feed-forward Neural Networks by Means of Binary Decision Diagrams"* [Bad09]. After decomposing a given network into its basic building blocks, a very compact description of their behaviour is extracted. This description is based on binary decision diagrams and allows for a simple re-combination of the intermediate results, which yields a logic program describing the behaviour of the network.

In Chapter 7, we discuss the embedding of first-order rules and the training of the resulting networks. We present different algorithms to construct connectionist systems for a given first-order logic program, such that the networks approximate the behaviour of the program. How to construct such networks has been completely unclear, until we showed a first idea in [BHW05]. An approach which allows for a better approximation has been published in *"A Fully Connectionist Model Generator for Covered First-Order Logic Programs"* [BHHW07] and *"Connectionist Model Generation: A First-Order Approach"* [BHH08]. This thesis contains (for the first time) all proofs showing the correctness of the approach and implementations of the necessary algorithms.

Some conclusions are drawn in Chapter 8, where we also discuss possible further work. Here we discuss again the challenge problems presented in Section 1.4, but now from a more informed perspective, including some solutions to them presented in this thesis.

This thesis presents the first uniform treatment of all above mentioned results. All ideas and algorithms are presented using the same notation and a Prolog implementation is provided for all algorithms. The same implementation has also been used to automatically create all examples presented throughout the thesis.

2 Preliminaries

... where all required concepts as well as the notation used throughout this thesis are introduced. First, we discuss some general notions and notations. In Section 2.2, we look into some mathematical results. In particular we recall notions like metrics and contractive functions. Afterwards, we present iterated function systems and discuss some of their properties. Then, we introduce the syntax and semantics of logic programs in Section 2.3 and binary decision diagrams in Section 2.4. Finally, in Section 2.5, we discuss artificial neural networks.

2.1 General Notions and Notations

As mentioned above, we start by introducing some notational conventions used throughout this thesis. If not mentioned otherwise, we make use of the following notations:

- *iff* means if and only if
- we use $\in_i$ to denote the i-th element of a given list
- we sometimes assume sets to be ordered and refer to them as *ordered set* (if no order is explicitly given, we use the lexicographic order, or the order in which the elements have been constructed) and, as for lists, we use $\in_i$ to denote the i-th element of this set with respect to this order
- we use $x^{[i]}$ to denote the i-th component of some vector x

We use the terms *(artificial) neural network* and *connectionist system* in this thesis as synonyms. If the distinction between biological or biologically plausible networks and artificial systems is necessary, we use *neural network* to refer to biological systems and *connectionist system* for artificial ones.

All algorithms presented below are shown as implementations in Prolog using SWI-Prolog [Wie09, www09c]. Figure 2.1 to 2.3 show the "framework" used for the implementation. Some specific operators used in the implementations as well as "syntactic sugar" resulting in better readable code are shown in Figure 2.1. This includes for example the implementation of $\in_i$ as introduced above. A major role play the operators `::=` and `:=` which should be read as "can be evaluated as". Their definitions are provided in Figure 2.2 and 2.3.

```
:- op(525,  fx, [:=, ::=]).
:- op(530, xfx, [:=, ::=]).
:- op(450, xfx, [=>]).
:- op(100, xfx, [@,..]).
:- op(450, xfx, [at,rest]).
:- op(450, xfy, ?).
:- op(440, xfy, [in]).
:- op(440, xfy, [union, subtract]).
:- op(440,  fx, unionover).

Elem in List :- !,
  RList := List,
  ( var(Elem) -> RElem = Elem
              ;  ( (Elem = (:=EElem)) -> RElem := EElem ; RElem = Elem ) ),
  member(RElem, RList).

Number in Number at 1 :- number(Number), !.

Elem in List at Index :- !,
  RList := List,
  ( var(Elem) -> RElem = Elem
              ;  ( (Elem = (:=EElem)) -> RElem := EElem ; RElem = Elem ) ),
  ( var(Index) -> RIndex = Index
              ;  ( (Index = (:=EIn)) -> RIndex := EIn ; RIndex = Index ) ),
  nth1(RIndex, RList, RElem).

Elem in List rest List2 :- !,
  RList := List,
  ( var(Elem) -> RElem = Elem
              ;  ( (Elem = (:=EElem)) -> RElem := EElem ; RElem = Elem ) ),
  select(RElem, RList, List2).

count_occurences(_Needle, [], 0).

count_occurences(Needle, [Needle2|Haystack], Number) :-
  copy_term(Needle, Needle2), !,
  count_occurences(Needle, Haystack, N),
  Number is N+1.

count_occurences(Needle, [_|Haystack], Number) :-
  count_occurences(Needle, Haystack, Number).
```

Figure 2.1: *Operators defined to obtain a more compact source code and some "syntactic sugar" yielding a better readable source code*

```
Expression := Expression :-
  var(Expression), !.

Result := #(Needle, Haystack) :- !,
  count_occurences(Needle, Haystack, Result).

Result := Type => Input :- !, parse(Type, Input, Result).

Result := [Low..High] :- !, RLow := Low, RHigh := High,
  numlist(RLow, RHigh, Result).

Result := {{ Var | Condition }} :- !,
  findall(Var, Condition, Result).

Result := { Var | Condition } :- !,
  findall(Var, Condition, ResultBag),
  sort(ResultBag, Result).

Result := { E1, E2 } :- !, E2R := { E2 }, is_list(E2R),
  evaluateList(Result, [E1|E2R]).

Result := { E } :- !, evaluateList(Result, [E]).

Result := unionover Sets :- !,
  RSets := Sets, is_list(RSets),
  Result := { Element | Set in RSets, RSet := Set,
                        is_list(RSet), Element in RSet }.

Result := Set1 union Set2 :-
  RSet1 := Set1, is_list(RSet1),
  RSet2 := Set2, is_list(RSet2), !,
  union(RSet1, RSet2, Result).

Result := Set1 subtract Set2 :-
  RSet1 := Set1, is_list(RSet1),
  RSet2 := Set2, is_list(RSet2), !,
  subtract(RSet1, RSet2, Result).

Result := size(List) :-
  RList := List, is_list(RList), !,
  length(RList, Result).

Result := last(List) :-
  RList := List, last(RList, Result).

Result := List :-
  is_list(List), !, evaluateList(Result, List).

Result := [Head|Tail] :-
  RHead := Head, RTail := Tail, List = [RHead|RTail],
  is_list(List), !, evaluateList(Result, List).

Result := min(List) :- RList := List, minimum(RList, Result), !.
Result := max(List) :- RList := List, maximum(RList, Result), !.
Result := sum(List) :- RList := List, sumlist(RList, Result), !.
```

Figure 2.2: *The definition for the "evaluate-to" operator*

```
Result := sort(List) :- Result := sort(List, compare), !.

Result := sort(List, Sorter) :- RList := List, is_list(RList), !,
  predsort(Sorter, RList, Result).

Result := Predicate :-
  Predicate =.. [N|Args], atom(N),
  evaluateList(RArgs, Args),
  append(RArgs, [Result], RRArgs),
  length(RRArgs, LRRArgs),
  current_predicate(N/LRRArgs), !,
  Predicate2 =.. [N|RRArgs], call(Predicate2).

Result := VecExpression :-
  VecExpression =.. [Op, Exp1, Exp2],
  RExp1 := Exp1, is_list(RExp1),
  RExp2 := Exp2, is_list(RExp2),
  vecOpVec(Op, RExp1, RExp2, Result), !.

Result := VecExpression :-
  VecExpression =.. [Op, Exp1, Exp2],
  RExp1 := Exp1, is_list(RExp1),
  RExp2 := Exp2, number(RExp2),
  vecOpNum(Op, RExp1, RExp2, Result), !.

Result := VecExpression :-
  VecExpression =.. [Op, Exp1, Exp2],
  RExp1 := Exp1, number(RExp1),
  RExp2 := Exp2, is_list(RExp2),
  numOpVec(Op, RExp1, RExp2, Result), !.

Result := MathExpression :-
  \+ var(MathExpression), MathExpression =.. [Functor|Arguments],
  current_arithmetic_function(Head),
  Head =.. [Functor|Arguments], !,
  evaluateArguments(EvaluatedArguments, Arguments),
  RealHead =.. [Functor|EvaluatedArguments], !,
  Result is RealHead.

Result := (Condition ? If : Else) :- !,
  RCondition := Condition,
  (RCondition -> (Result := If) ; (Result := Else)).

Expression := Expression :- !.

evaluateList([], []).
evaluateList([Value|Values], [:= Argument|Arguments]) :-
  \+ var(Argument), !, Value := Argument, evaluateList(Values, Arguments).
evaluateList([Value|Values], [::= Argument|Arguments]) :-
  \+ var(Argument), !, Value ::= Argument, evaluateList(Values, Arguments).
evaluateList([Value|Values], [Value|Arguments]) :-
  evaluateList(Values, Arguments).

evaluateArguments([], []).
evaluateArguments([Variable|Values], [Variable|Arguments]) :-
  var(Variable), !, evaluateArguments(Values, Arguments).
evaluateArguments([Value|Values], [Argument|Arguments]) :-
  \+ var(Argument), !, Value := Argument, evaluateArguments(Values, Arguments).
```

Figure 2.3: *The definition for the "evaluate-to" operator, ctd.*

2.2 Metric Spaces, Contractive Functions and Iterated Function Systems

In this section, we repeat some basic mathematical notions and ideas which are relevant in the sequel. First, we introduce metric spaces. Afterwards, we discuss continuous and contractive functions in some detail. Contractive mappings play an important role throughout this thesis. In Section 2.2.3 they are used while defining *Iterated Function Systems*, in Section 2.3 to show the existence of least models for certain logic programs, and they are also used to show the feasibility of the first-order approach as presented in Chapter 7. We follow in principle [Bar93], but all notions presented here should be covered in any textbook on elementary calculus.

2.2.1 Metric Spaces

A set X together with a metric $m : X \times X \to \mathbb{R}$, is called a *metric space*. The metric (also called distance measure) m defines a notion of distance between elements of X. To be a metric, this function m must fulfil certain conditions: the *identity of indiscernibles*, *symmetry* and the *triangle inequality* as spelled out in the following definition.

Definition 2.2.1 (Metric, Metric Space) *Let X be some set. A function $m : X \times X \to \mathbb{R}$ is called a* metric *on X, if the following conditions hold for all $x, y, z \in X$:*

$$\begin{array}{lll} (i) & m(x,y) = 0 \text{ iff } x = y & \textit{(identity of indiscernibles)} \\ (ii) & m(x,y) = m(y,x) & \textit{(symmetry)} \\ (iii) & m(x,z) \leq m(x,y) + m(y,z) & \textit{(triangle inequality)} \end{array}$$

We call (X, m) a metric space.

Two examples of metric spaces, frequently used in the sequel, are presented in Definition 2.2.2 and 2.2.3. Some further metric spaces are shown in Example 2.2.1.

Definition 2.2.2 (Euclidean Metric, Euclidean Space) *Let $d \geq 1$ be a natural number and $\mathbb{R}^d$ be resulting set of d-dimensional vectors over real numbers. The function*

$$\begin{aligned} m_{\mathfrak{e}} :& \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \\ & (x_1, \ldots, x_d, y_1, \ldots, y_n) \mapsto \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} \end{aligned}$$

is called the Euclidean metric *over $\mathbb{R}^d$ and $(\mathbb{R}^d, m_{\mathfrak{e}})$ the d-dimensional* Euclidean Space.

Definition 2.2.3 (Maximum Metric) *Let $\mathbb{R}^d$ be the set of d-dimensional vectors over real numbers for $d \geq 1$. The function*

$$\begin{aligned} m_{\mathfrak{m}} :& \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \\ & (x_1, \ldots, x_d, y_1, \ldots, y_n) \mapsto \max_{1 \leq i \leq d} |x_i - y_i| \end{aligned}$$

is called the maximum metric *over $\mathbb{R}^d$.*

Two metric spaces (X_1, m_1) and (X_2, m_2) are said to be *equivalent* (also called similar), if the distances between elements are approximately the same. This can be made precise if there is a function $\iota : X_1 \to X_2$ and two constants e_1 and e_2 such that $e_1 \cdot m_1(x, y) \leq m_2(\iota(x), \iota(y)) \leq e_2 \cdot m_1(x, y)$. I.e., the distances in both spaces are linked to each other.

Example 2.2.1 (Some Metric Spaces) Some often used metric spaces:

a) The real numbers $\mathbb{R}$ (rational numbers $\mathbb{Q}$) together with the Euclidean metric.

b) The d-dimensional real vectors with the maximum metric.

c) The prefix metric on Strings (inverse of the length of the common prefix).

d) The Hausdorff metric between non-empty compact subsets (discussed in Section 2.2.3).

Definition 2.2.4 (Equivalent Metric Spaces) *The metric spaces (X_1, m_1) and (X_2, m_2) are said to be equivalent with respect to $0 < e_1 < e_2 < \infty \in \mathbb{R}$ and $\iota : X_1 \to X_2$ if for all $x, y \in S_1$ the following holds:*

$$e_1 \cdot m_1(x, y) \leq m_2(\iota(x), \iota(y)) \leq e_2 \cdot m_1(x, y)$$

Using the notion of distance as provided by a metric, we can define convergence formally. A sequence $x_1, x_2, \ldots$ on (X, m) is said to converge to some point $x \in X$, if the distance between the x_i and x is decreasing over time.

Definition 2.2.5 (Convergent Sequence, Limit Point) *Let (X, m) be a metric space. Then the sequence $x_1, x_2, \ldots$ is said to* converge *to some point $x \in X$, if for every $\varepsilon > 0$ there is some n_ε such that $m(x_i, x) < \varepsilon$ for all $i \geq n_\varepsilon$. We call x the* limit point *of the sequence.*

Example 2.2.2 ((Non)-Converging Sequences) Some sequences and their limit points:

a) $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$ does converge on $(\mathbb{R}, m_{\mathfrak{E}})$ to 0.

b) $1, 2, 3, 4, \ldots$ does not converge on e.g., $(\mathbb{N}, m_{\mathfrak{E}})$.

c) $\left(1 + \frac{1}{1}\right)^1, \left(1 + \frac{1}{2}\right)^2, \left(1 + \frac{1}{3}\right)^3, \left(1 + \frac{1}{4}\right)^4, \ldots$ does converge to $e = 2.718\ldots$ on $(\mathbb{R}, m_{\mathfrak{E}})$.

Definition 2.2.6 (Cauchy Sequence) *Let (X, m) be a metric space. A sequence $x_1, x_2, \ldots$ is called* Cauchy sequence *or* Cauchy*, if for any $\epsilon > 0$ there is an index n_ε such that for all $i, j > n_\varepsilon$ we find $m(x_i, x_j) < \varepsilon$.*

Because the distance between elements of a Cauchy sequence gets smaller with increasing index, we find that the sequence is approaching some point. But it does not necessarily converge, because it might approach some point x which is not an element of X. E.g., the sequence c) in Example 2.2.2 is converging on $(\mathbb{R}, m_{\mathfrak{E}})$ to the constant e, but it is not converging on $(\mathbb{Q}, m_{\mathfrak{E}})$, because even though all elements $\left(1 + \frac{1}{i}\right)^i$ are rational, e itself is not. If for some metric space all cauchy sequences converge, we call it *complete*. Such a complete space does not have any "holes", i.e., there are no points missing. For example the real numbers are complete, while the rational numbers are not.

Definition 2.2.7 (Complete Metric Space) *Let (X, m) be a metric space. If every Cauchy sequence on it converges to some point $x \in X$, we call (X, m) a* complete metric space*.*

Definition 2.2.8 (Closed) *Let (X, m) be a metric space. A subset $S \subseteq X$ is called* closed*, if every converging sequence $s_1, s_2, \ldots$ with $s_i \in S$ converges to some point $s \in S$.*

We call a subset S of a metric spaces *bounded*, if it is contained in some ball of finite radius with respect to the underlying metric. A subset is called totally bounded, if there is a so-called finite cover of it. I.e., if we can cover it with a finite set of balls for any radius.

Definition 2.2.9 (Bounded) *Let (X, m) be a metric space. A subset $S \subseteq X$ is called* bounded*, if there is a number $r \in \mathbb{R}$, such that for all $s_1, s_2 \in S$ we find $m(s_1, s_2) \leq r$.*

Definition 2.2.10 (Totally Bounded) *A subset $S \subseteq X$ of a metric space (X, m) is called* totally bounded*, if for all $\varepsilon > 0$ there is a finite set $S_\varepsilon \subseteq S$ such that $m(x, s_i) < \varepsilon$ holds for all $x \in S$ and some $s_i \in S_\varepsilon$.*

Totally bounded and closed sets are called *compact*. Intuitively, a compact set contains its border. I.e., all the limit points of converging sequences, and the distances between all elements are restricted.

Definition 2.2.11 (Compact) *Let (X, m) be a complete metric space. We call $S \subseteq X$* compact*, if S is totally bounded and closed.*

Compact sets are used below to define iterated function systems. They also play an important role in the theory of connectionist systems, because continuous functions over compact sets can be approximated arbitrarily well using certain connectionist systems.

2.2.2 Continuous and Contractive Functions

Using the notion of distance as introduced in Definition 2.2.1, we can define continuous functions. A function $f : X \to Y$ on the metric spaces (X, m_X) and (Y, m_Y) is called *continuous* if small changes in the input cause only small changes in the output. More precisely, if for all $\varepsilon > 0$ there is some $\delta > 0$ such that $m_X(x, y) < \delta$ implies $m_Y(f(x), f(y)) < \epsilon$ for all $x, y \in X$, then we call f uniformly continuous.

Definition 2.2.12 ((Uniformly) Continuous) *Let (X, m_X) and (Y, m_Y) be metric spaces. A function $f : X \to Y$ is called* continuous at x *(with respect to m_X and m_Y), iff for every point $x \in X$ and every $\epsilon > 0$ there is a $\delta > 0 \in \mathbb{R}$ such that $m_X(x, y) < \delta$ implies $m_Y(f(x), f(y)) < \epsilon$ for all $y \in X$. A mapping is called* continuous *if it is continuous at every point $x \in X$. It is furthermore called* uniformly continuous *if there is a δ for every $\epsilon > 0$ such that for all $x, y \in X$ we find $m_X(x, y) < \delta$ implies $m_Y(f(x), f(y)) < \epsilon$.*

We usually use the term continuous function to denote uniformly continuous functions. If there is a linear dependency between ε and δ, we obtain the notion of *Lipschitz continuity* as defined below. Please note that Lipschitz continuity implies continuity with $\delta = \varepsilon / L$.

Definition 2.2.13 (Lipschitz Continuous) *Let $f : X \to Y$ be a function between the two metric spaces (X, m_X) and (Y, m_Y). f is called* Lipschitz continuous*, if there is a constant $0 \leq L$ such that we find $m_Y(f(x), f(y)) \leq L \cdot m_X(x, y)$ for all $x, y \in X$. The constant L is called* Lipschitz constant *for f.*

Lipschitz continuity of some function $f : X \to X$ is preserved if it is embedded into a space Y using a bijective mapping $\iota : X \to Y$, and under the condition that (X, m_X) and (Y, m_Y) ate equivalent with respect to ι, e_1 and e_2.

Lemma 2.2.14 *Let (X, m_X) and (Y, m_Y) be equivalent with respect to $\iota : X \to Y$, e_1 and e_2. Let $f : X \to X$ be Lipschitz continuous with constant L_f and let ι be bijective. Then we find $g : Y \to Y : x \mapsto \iota(f(\boldsymbol{\iota^{-1}}(x)))$ to be Lipschitz continuous with constant $L_g = \frac{e_2}{e_1} \cdot L_f$.*

Proof We need to show $m_Y(g(x), g(y)) \leq L_g \cdot m_Y(x, y)$ for all $x, y \in Y$.

$$\begin{aligned}
m_Y(g(x), g(y)) &= m_Y(\iota(f(\boldsymbol{\iota}^{-1}(x))), \iota(f(\boldsymbol{\iota}^{-1}(y)))) && \% \text{ by definition of } g \\
&\leq e_2 \cdot m_X(f(\boldsymbol{\iota}^{-1}(x)), f(\boldsymbol{\iota}^{-1}(y))) && \% \text{ equivalence of } (X, d_X) \text{ and } (Y, d_Y) \\
&\leq e_2 \cdot L_f \cdot m_X(\boldsymbol{\iota}^{-1}(x), \boldsymbol{\iota}^{-1}(y)) && \% \text{ by Lipschitz continuity of } f \\
&\leq \frac{e_2}{e_1} \cdot L_f \cdot m_Y(x, y) && \% \text{ equivalence of } (X, d_X) \text{ and } (Y, d_Y) \\
&\leq L_g \cdot m_Y(x, y) && \% \text{ by definition of } L_g
\end{aligned}$$ □

Lipschitz continuous functions with a constant $L < 1$ are called contractive. This is due to the fact that we find $m_Y(g(x), g(y)) < m_Y(x, y)$ for all $x, y \in Y$. A contractive mapping on some metric space possesses a unique fixed point as shown in *Banach's contraction mapping theorem* below.

Definition 2.2.15 (Contractive) *A Lipschitz continuous function f with Lipschitz constant L is called contractive if $L < 1$.*

Theorem 2.2.16 (Banach's Contraction Mapping Theorem) *Let $f : X \to X$ be a contractive mapping on the complete metric space (X, m). Then, f possesses exactly one fixed point $x \in X$ and for any point $y \in X$ we find that the sequence $y, f(y), f^2(y), \ldots$ converges to x.*

Proof (sketch) This theorem can be proven, by first showing that the sequence is Cauchy and hence converging to some $x \in X$ because (X, m) is complete. From the contractivity of f we can conclude that this x is a fixed point of f. The uniqueness of x can be shown by deriving a contradiction while assuming that there are two such points. □

As already mentioned above, contractive mappings play an important role throughout this thesis, because they define unique fixed points. For example in the following section Banach's theorem ensures the uniqueness of the attractor of so-called iterated function systems.

2.2.3 Iterated Function Systems

An *iterated function system* (IFS) consists of a complete metric space and a set of contractive functions defined on it. As already mentioned above we follow [Bar93]. Before showing examples, we define some required notions, namely *iterated function systems* and their corresponding *attractors*.

Definition 2.2.17 (Iterated Function System) *For some complete metric space (X, d) and a finite set of contractive mappings $\Omega = \{\omega_1, \ldots, \omega_n\}$ with $\omega_i : X \to X$, we call $\langle (X, d), \Omega \rangle$ an* iterated function system.

Let (X, d) be the complete metric space of a given IFS and let $\mathcal{H}(X)$ denote the set of all non-empty compact subsets of X. Then we define the *Hausdorff distance* h_d on $\mathcal{H}(X)$ as follows:

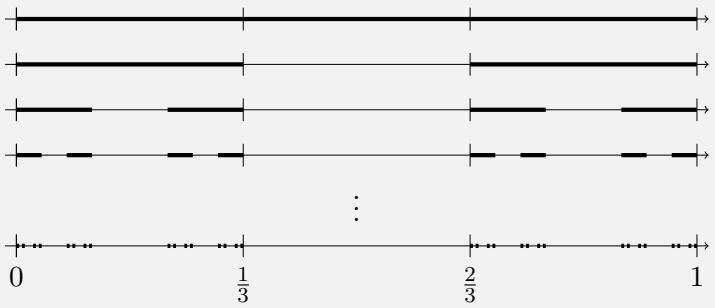
$$\begin{aligned}
&h_d : \mathcal{H}(X) \times \mathcal{H}(X) \to \mathbb{R} : (A, B) \mapsto \max(h'_d(A, B), h'_d(B, A)) \quad \text{with} \\
&h'_d : \mathcal{H}(X) \times \mathcal{H}(X) \to \mathbb{R} : (A, B) \mapsto \max_{a \in A} \{ \min_{b \in B} d(x, b) \}
\end{aligned}$$

As shown for example in [Bar93], the set $\mathcal{H}(X)$ together with h_d, is a complete metric space. Moreover, the set-extension[1] of contractive mappings on (X, d) is contractive on $(\mathcal{H}(X), h_d)$. Therefore, we can conclude that $\Omega : \mathcal{H}(X) \to \mathcal{H}(X) : A \mapsto \cup_{\omega\in\Omega}\omega(A)$ is also contractive and hence possesses a unique fixed point. This fixed point is called the *attractor* of the iterated function system.

Definition 2.2.18 (Attractor) *Let $\langle (X, d), \Omega\rangle$ be an iterated function system. Let $\mathcal{H}(X)$ and h_d be as introduced above and let $\Omega : \mathcal{H}(X) \to \mathcal{H}(X)$ be defined as $\Omega(A) = \cup_{\omega\in\Omega}\omega(A)$. The unique fixed point $A \in \mathcal{H}(X)$ is called the* attractor *of $\langle (X, d), \Omega\rangle$.*

We can approximate the attractor of a given IFS by starting with an arbitrary compact subset and applying the mappings repeatedly to it. Banach's theorem ensures that the images of this process approximate the attractor.

Example 2.2.3 Let $(\mathbb{R}, m_{\mathfrak{E}})$ be the real line equipped with the usual metric and let $\Omega = \{\omega_1, \omega_2\}$ with $\omega_1 : x \mapsto \frac{x}{3}$ and $\omega_2 : x \mapsto \frac{x}{3} + \frac{2}{3}$. The first 4 iterations of the construction of the attractor starting with the unit interval and the attractor itself are shown below:



Please note, that the result, i.e., the attractor, of this IFS is the usual Cantor space [Bar93].

We use iterated function systems to denote their attractors. As already mentioned in Section 1.2.2, there are also links between IFSs and logic programs. The graph of the associated T_P-operator is the attractor of a suitably set up IFS. This finding lead to a novel approach to relate logic programs and connectionist system, because the IFSs can be used to bridge the gap between the world of discrete interpretations and the continuous real numbers.

All proofs and much more details on iterated function systems can be found in Michael Barnsley's marvellous book [Bar93]. In the following sections, we introduce logic programs, binary decision diagrams and finally connectionist systems.

2.3 Logic Programs

Logic programs are a well known formalism for the representation of knowledge, in particular for if-then rules. The syntax is introduced in Section 2.3.1, and the semantics in Section 2.3.2.

This section is not meant to give a complete introduction into the field of logic programs, but rather to introduce all notions required throughout this thesis. We follow in principle [Llo87], which contains all details and provides a nice introduction into the area of logic programming. Of particular interest is the T_P-operator associated with every logic program. Furthermore, we discuss so-called level mappings in some detail and a metric on interpretations needed in later parts of this thesis.

[1] By set-extension of some function $f : X \to X$ we mean $f : \mathcal{P}(X) \to \mathcal{P}(X) : f(A) = \{f(a) \mid a \in A\}$.

2.3.1 Syntax

A logic program is a list of if-then rules over some given language. Each rule consists of a conjunction of preconditions and a single conclusion. After defining first-order languages, we define logic programs formally and discuss some basic properties important for later parts of this thesis.

Definition 2.3.1 (First-Order Language) *Let $\mathcal{F} = \{f_1/a_1, \ldots, f_n/a_n\}$ be a set of function symbols f_i with arities a_i and let $\mathcal{R} = \{r_1/b_1, \ldots, r_m/b_m\}$ be a set of relation symbols r_i with arities b_i. Then we call $\mathcal{L} = \langle \mathcal{R}, \mathcal{F} \rangle$ a* first-order language.

Example 2.3.1 Consider the language containing the functions $0/0$ and $s/1$ and the relations $e/1$ and $o/1$. Intuitively they may denote the constant 0, the successor function s and the properties that something is an even or an odd number.

A logic program is built from clauses which in turn are built from atomic formulae, i.e., formulae of the form $r(t_1, \ldots, t_a)$ for some $r/a \in \mathcal{R}$ with t_i being terms built over variables and functions from $\mathcal{F}$. Every clause consists of a head, which is an atom, and a list of possibly negated atoms constituting the body. We write clauses in the form

$$H \leftarrow L_1 \wedge \ldots \wedge L_n$$

with H being the head and $L_1 \wedge \ldots \wedge L_n$ being the body.

Definition 2.3.2 (Logic Program) *A* logic program *is a list of clauses of the form $H \leftarrow L_1, \ldots, L_n$, where n can differ for each clause. H is an atom and called the head of the clause. $L_1 \wedge \ldots \wedge L_n$ is called the body, and each L_i is a literal, i.e., an atom or negated atom.*

We use $C \in_i P$ to denote the i-th clause in P. While talking about source codes we use the Prolog notation for logic programs, in which a clause $H \leftarrow L_1 \wedge \ldots \wedge L_n$ is written as $\mathtt{H} :- \mathtt{L_1}, \ldots, \mathtt{L_n}$. Example 2.3.2 to 2.3.4 show some logic programs which serve as running example throughout the thesis.

If the underlying language contains relation symbols of arity 0 only, it is said to be propositional. A program built on-top of it is called *propositional logic program*. The 0-ary relation symbols of a propositional language are usually called *propositional variables*.

Example 2.3.2 (A Small Propositional Program) The intended meaning of the single clauses is given as "*% comment*" on the right.

$$\begin{array}{ll} a. & \%\ a\ \textit{is always true, i.e. it is a fact} \\ b \leftarrow a. & \%\ b\ \textit{is true if}\ a\ \textit{is true} \\ c \leftarrow \neg a. & \%\ c\ \textit{is true if}\ a\ \textit{is false} \\ c \leftarrow a \wedge b. & \%\ c\ \textit{is true if}\ a\ \textit{and}\ b\ \textit{are true} \end{array}$$

Definition 2.3.3 (Local Variable) *A variable X occurring in the body but not in the head of a clause is called* local variable.

Example 2.3.3 (Natural Numbers) The following program defines natural numbers.

$$\begin{array}{ll} nat(0). & \%\ 0\ \textit{is a natural number} \\ nat(s(X)) \leftarrow nat(X). & \%\ \textit{If } X\ \textit{is a natural number} \\ & \%\ \textit{then the successor } s(X)\ \textit{as well.} \end{array}$$

Example 2.3.4 (Even and Odd Numbers) The even and odd numbers can be formalised as follows:

$$\begin{array}{ll} even(0). & \%\ 0\ \textit{is an even number} \\ even(s(X)) \leftarrow odd(X). & \%\ \textit{If } X\ \textit{is an odd number} \\ & \%\ \textit{then the successor } succ(X)\ \textit{is even.} \\ odd(X) \leftarrow \neg even(X). & \%\ X\ \textit{is odd, if it is not even.} \end{array}$$

Definition 2.3.4 (Covered Logic Program) *A logic program without local variables is called* covered.

2.3.2 Semantics

The set of all terms constructible from a set of function symbols is called *Herbrand universe*. Formally, it is defined to be the closure of the function composition over all functions contained in $\mathcal{F}$. It can be constructed bottom up, starting with all constants, i.e., function symbols with arity 0. To all those terms, all functions are applied and the result added to the set of terms, etc.

Definition 2.3.5 (Herbrand Universe) *Let $\mathcal{L} = \langle \mathcal{R}, \mathcal{F} \rangle$ be some first-order language. The* Herbrand Universe *is the smallest set $U_\mathcal{L}$ for which $f/a \in \mathcal{F}$ and $t_1, \ldots, t_a \in U_\mathcal{L}$ implies that $f(t_1, \ldots, t_a) \in U_\mathcal{L}$ holds.*

Applying all relation symbols to all possible combinations of elements of the Herbrand universe yields the *Herbrand base*.

Definition 2.3.6 (Herbrand Base) *Let $\mathcal{L} = \langle \mathcal{R}, \mathcal{F} \rangle$ be some first-order language and let $U_\mathcal{L}$ be the corresponding Herbrand Universe. The* Herbrand Base *is the smallest set $B_\mathcal{L}$ for which $r/a \in \mathcal{R}$ and $t_1, \ldots, t_a \in U_\mathcal{L}$ implies that $r(t_1, \ldots, t_a) \in B_\mathcal{L}$ holds.*

Reconsidering the language from Example 2.3.1 we obtain the Herbrand universe and base given in Example 2.3.5.

Example 2.3.5 For the language from Example 2.3.1, i.e., $\mathcal{L} = \langle \{0/0, s/1\}, \{e/1, o/1\} \rangle$, we find:

$$\begin{aligned} U_\mathcal{L} &= \{0, s(0), s(s(0)), s(s(s(0))), s(s(s(s(0)))), \ldots\} \\ B_\mathcal{L} &= \{e(0), e(s(0)), e(s(s(0))), \ldots\} \cup \{o(0), o(s(0)), o(s(s(0))), \ldots\} \end{aligned}$$

Using the Herbrand universe associated with a given language, we can define ground terms and atoms as usual. They are obtained by replacing variables consistently by all ground terms, i.e., by all elements of the Herbrand universe.

Definition 2.3.7 (Ground Program ground(P)**)** *Let $\mathcal{L} = \langle \mathcal{R}, \mathcal{F} \rangle$ be some first-order language, let $U_\mathcal{L}$ be the associated Herbrand universe and let P be a program defined over $\mathcal{L}$. Then the* ground program ground(P) *is defined to be the set of all ground instances obtained from clauses in P.*

Obviously, the ground version of most first-order programs is infinite. This follows from the infiniteness of the corresponding Herbrand universe. As soon as the language contains at least one function symbols of arity 0 and one of higher arity, we find the Herbrand universe to be infinite. Therefore, the ground version of such a program is infinite as well.

The Single Step Operator

The knowledge represented in a logic program P can be captured by the *single step operator* T_P (aslo called *immediate consequence operator, or just consequence operator*). It maps interpretations to interpretations and propagates truth along the clauses of the program. This operator is defined as follows:

Definition 2.3.8 (T_P-Operator) *Let P be a logic program defined over some first-order language $\mathcal{L}$, let $B_\mathcal{L}$ be the corresponding Herbrand base and let $\mathcal{I}_\mathcal{L} := \mathcal{P}(B_\mathcal{L})$ be the set of all interpretations. The T_P-operator associated with the program P is defined as follows:*

$$T_P : \mathcal{I}_\mathcal{L} \to \mathcal{I}_\mathcal{L} : I \mapsto \{A \mid (A \leftarrow Body) \in \text{ground}(P) \text{ with } I \models Body\}$$

Obviously, this definition is applicable to propositional programs as well as to first-order programs. But the T_P-operator associated with a propositional program P can be re-defined as follows:

Definition 2.3.9 (Propositional T_P-Operator) *Let P be a propositional logic program defined over some set of propositional variables $\mathcal{V}$ and let $\mathcal{I}_\mathcal{L}$ denote the set of all interpretations, i.e. $\mathcal{I}_\mathcal{L} = \mathcal{P}(\mathcal{V})$. The T_P-operator associated with the program P is defined as follows:*

$$T_P : \mathcal{I}_\mathcal{L} \to \mathcal{I}_\mathcal{L} : I \mapsto \{A \mid (A \leftarrow Body) \in P \text{ with } I \models Body\}$$

Example 2.3.6 and 2.3.7 show the T_P-operators associated with the logic programs from Example 2.3.2 and 2.3.3.

Example 2.3.6 Considering the program from Example 2.3.2, we find the T_P-operator to be as follows:

$\mathbf{I}$	$\mathbf{T_P(I)}$
$\{\}$	$\{a, c\}$
$\{a\}$	$\{a, b\}$
$\{b\}$	$\{a, c\}$
$\{a, b\}$	$\{a, b, c\}$

$\mathbf{I}$	$\mathbf{T_P(I)}$
$\{c\}$	$\{a, c\}$
$\{a, c\}$	$\{a, b\}$
$\{b, c\}$	$\{a, c\}$
$\{a, b, c\}$	$\{a, b, c\}$

Please not that in Prolog programs, the negation symbol $\neg$ is usually interpreted as a "negation as failure". I.e., the atom $\neg a$ is considered to be true if and only if the Prolog interpreter fails to prove a. Looking at the definition of the T_P-operator, we find that $I \models B$ must be fulfilled such that the head of the clause with body B is included in the result of the T_P-operator. And for some Herbrand interpretation I we find $I \models \neg a$ if and only if $a \notin I$. I.e., while using the T_P-operator, the negation symbol $\neg$ is interpreted as classical negation.

Example 2.3.7 The first three iterations of the T_P-operator associated to the *natural numbers program* from Example 2.3.3:

$$\{\} \overset{T_P}{\mapsto} \{nat(0)\} \overset{T_P}{\mapsto} \{nat(0), nat(s(0))\} \overset{T_P}{\mapsto} \{nat(0), nat(s(0)), nat(s(s(0)))\}$$

Level Mappings

Using so-called *level mappings* we can bridge the gap between the discrete space of interpretations and the continuous space of real numbers. How this can be done is discussed in Section 7.2. Usually those mappings are used while discussing the termination of the iteration of the T_P-operator. We introduce (multi-dimensional) level mappings first and later define a metric on interpretations.

Definition 2.3.10 (Level Mapping) *Let $B_\mathcal{L}$ be some Herbrand base. A function $|\cdot| : B_\mathcal{L} \to \mathbb{N}^+$ is called a* level mapping *for $B_\mathcal{L}$. For $A \in B_\mathcal{L}$ and $|A| = l$, we call l the* level *of A. Furthermore, we define $|\neg A| := |A|$.*

Example 2.3.8 A possible level mapping for the Herbrand base from Example 2.3.5 (corresponding to the program shown in Example 2.3.4) can be defined as follows:

$$|\cdot| : B_\mathcal{L} \to \mathbb{N}^+ \quad \text{with} \quad \begin{cases} e(s^n(0)) \mapsto 2 \cdot n + 1 \text{ and} \\ o(s^n(0)) \mapsto 2 \cdot n + 2. \end{cases}$$

A possible level mapping for the program from Example 2.3.4 is shown in Example 2.3.8. Even though the following definition just repeats the usual notion of injectivity and bijectivity, we restate them here because both play an important role throughout the thesis.

Definition 2.3.11 *Let $|\cdot|$ be some level mapping for $B_\mathcal{L}$. We call $|\cdot|$ to be injective, iff $|A| = |B|$ implies $A = B$ for all $A, B \in B_\mathcal{L}$. We call $|\cdot|$ to be bijective iff it is injective and there exists a A for each $n \in \mathbb{N}^+$ such that $|A| = n$.*

Usually, level mappings are defined as above, but throughout this thesis we need a more flexible definition. Instead of mapping atoms to natural numbers, we map them to pairs of natural numbers as follows:

Definition 2.3.12 (d-Dimensional Level Mapping) *Let $d \in \mathbb{N}^+$ and $B_\mathcal{L}$ be some Herbrand base. Then we call $\|\cdot\| : B_\mathcal{L} \to (\mathbb{N}^+, \{1, \ldots, d\})$ a* d-dimensional level mapping *for $B_\mathcal{L}$. For $A \in B_\mathcal{L}$ and $\|A\| = (l_A, d_A)$, we call l_A and d_A the* level *and* dimension *of A respectively. Again, we define $\|\neg A\| := \|A\|$. Furthermore, we use $\|A\|_L$ and $\|A\|_D$ to denote the level and dimension of A, respectively.*

Example 2.3.9 shows a possible 2-dimensional level mapping for the Herbrand base from Example 2.3.5 Using level mappings as just introduced, we can define a restricted T_P-operator as follows:

Definition 2.3.13 (Restricted Semantic Operator $\lfloor T_P \rfloor_n$) *Let $n > 0$, $\|\cdot\|$ and T_P be given. Then we define the* restricted T_P-operator *as follows:*

$$\lfloor T_P \rfloor_n(I) = \{A \in B_\mathcal{L} \mid A \in T_P(I) \text{ and } \|A\|_L \leq n\}$$

Example 2.3.9 A possible 2-dimensional level mapping for the Herbrand base from Example 2.3.5 (corresponding to the program shown in Example 2.3.4):

$$\|\cdot\| : B_{\mathcal{L}} \to (\mathbb{N}^+, \{1,2\}) \quad \text{with} \quad \begin{cases} e(s^n(0)) \mapsto (n+1, 1) \text{ and} \\ o(s^n(0)) \mapsto (n+1, 2). \end{cases}$$

All *even*-atoms are mapped to the first dimension, and the *odd*-atoms to the second.

In Section 7.4 we also need to restrict the grounded version of a given program to those clauses for which the head has a level smaller than a given n. We refer to this subset of ground(P) as $\lfloor P \rceil_n$. Please note that this set can be infinite, if we construct it for non-covered programs, but as soon as the underlying P is covered, we can conclude that $\lfloor P \rceil_n$ is finite.

Definition 2.3.14 (Restricted Program $\lfloor P \rceil_n$) *Let P be a covered logic program over $B_{\mathcal{L}}$, let $\|\cdot\|$ be some d-dimensional level mapping and let $n > 0$. Then $\lfloor P \rceil_n \subseteq$ ground(P) is defined as follows:*

$$\lfloor P \rceil_n = \{H \leftarrow L_1 \wedge \ldots \wedge L_c \mid H \leftarrow L_1 \wedge \ldots \wedge L_c \in \text{ground}(P) \text{ and } \|H\|_L \leq n\}$$

The following lemma states the finiteness of $\lfloor P \rceil_n$ for a covered program P. It follows from the fact that the number of possible head atoms is bounded due to the bijectivity of the underlying level mapping. From the fact that the heads are limited and that there are no local variables, we can conclude that the number of different clauses is limited as well and hence that the restricted program is finite.

Lemma 2.3.15 *Let P be an covered logic program. Then $\lfloor P \rceil_n$ is finite.*

Metrics on Interpretations

Using level mappings as defined above, we define the metric $m_{\mathcal{L},b}$ as follows.

Definition 2.3.16 *Let $\mathcal{L}$ be some first-order language, let $B_{\mathcal{L}}$ be the corresponding Herbrand base and let $\mathcal{I}_{\mathcal{L}} = \mathcal{P}(B_{\mathcal{L}})$ be the set of all interpretations as above. For $b \geq 2$ we define $m_{\mathcal{L},b} : \mathcal{I}_{\mathcal{L}} \times \mathcal{I}_{\mathcal{L}} \to \mathbb{R}$ as follows:*

$$m_{\mathcal{L},b} : \mathcal{I}_{\mathcal{L}} \times \mathcal{I}_{\mathcal{L}} \to \mathbb{R}$$
$$(I, J) \mapsto \begin{cases} 0 & \text{if } I = J \\ b^{-l} & \text{if } I \text{ and } J \text{ differ on some atom of level } l \\ & \text{and agree on all atoms of smaller level} \end{cases}$$

Proposition 2.3.17 *$m_{\mathcal{L},b}$ as introduced above is a metric and the space $(\mathcal{I}_{\mathcal{L}}, m_{\mathcal{L},b})$ is compact.*

Proof $m_{\mathcal{L},b}$ is by definition symmetric and yields 0 if and only if applied to the same interpretation. I.e., it remains to show that the triangle inequality is also fulfilled. To show that $m_{\mathcal{L},b}(I,K) \leq m_{\mathcal{L},b}(I,J) + m_{\mathcal{L},b}(J,K)$ holds for all $I, J, K \in \mathcal{I}_{\mathcal{L}}$, we assume without loss of generality $m_{\mathcal{L},b}(I,J) = b^{-m}$, $m_{\mathcal{L},b}(J,K) = b^{-n}$ and $m < n$. Because I and J agree on all atoms with level smaller m and $m < n$, we find that J and K must agree on those atoms as well. Therefore, also I and K must agree on those atoms and we find $m_{\mathcal{L},b}(I,K) = b^{-m}$. Consequently, we conclude $m_{\mathcal{L},b}(I,K) = \max(m_{\mathcal{L},b}(I,J), m_{\mathcal{L},b}(J,K)) \leq m_{\mathcal{L},b}(I,J) + m_{\mathcal{L},b}(J,K)$ and, hence, that $m_{\mathcal{L},b}$ is a metric.

To prove the compactness we need to shoe that $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ is totally bounded and closed. The total boundedness can be shown by constructing the finite cover S_ε from Definition 2.2.10 as follows: Let $\varepsilon > 0$ be given. Then we select an n such that $b^{-n} < \varepsilon$ and compute S_ε as follows:

$$S_\varepsilon = \mathcal{P}\left(\{A \in B_\mathcal{L} \mid \|A\|_L \leq n\}\right)$$

Obviously this set is finite and we find for all $I \in \mathcal{I}_\mathcal{L}$ that $m_{\mathcal{L},b}(I, s_i) < \varepsilon$ for some $s_i \in S_\varepsilon$, namely for the s_i that agrees with I on all atoms A with $\|A\|_L \leq n$.

It remains to show that $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ is closed. For this we assume a converging sequence $s_1, s_2, \ldots$ with $s_i \in \mathcal{I}_\mathcal{L}$ and construct an interpretation $I \in \mathcal{I}_\mathcal{L}$ such that I is the limit point of this sequence. Due to the fact that the sequence is converging to some x, we can conclude that for every $\varepsilon > 0$ there is some n_ε such that $m_{\mathcal{L},b}(s_i, x) < \varepsilon$ for all $i \geq n_\varepsilon$ and furthermore, that $m_{\mathcal{L},b}(s_i, s_j) < 2\varepsilon$ for all $i, j \geq n_\varepsilon$. I.e., all s_i for $i \geq n_\varepsilon$ agree on all atoms with level smaller l such that $b^{-l} \leq \varepsilon$. We construct I as follows:

$$I := \{A \in B_\mathcal{L} \mid A \in s_i \text{ for all } i \geq n_\varepsilon \text{ with } \varepsilon := b^{-\|A\|_L}\}$$

and find that I is the limit point of the sequence $s_1, s_2, \ldots$, and therefore, we can conclude that $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ is compact. □

$(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ is

Remark 2.3.18 *Note that $m_{\mathcal{L},b}(I, K) \leq \max(m_{\mathcal{L},b}(I, J), m_{\mathcal{L},b}(J, K))$ holds, which turns $m_{\mathcal{L},b}$ into an ultra metric on $\mathcal{I}_\mathcal{L}$.*

Due to the way the metric $m_{\mathcal{L},b}$ is constructed, we can put the following bound on the contractivity factor of a contractive T_P-operator.

Lemma 2.3.19 *Let T_P be contractive on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ for base b with contractivity factor c. Then we find $c \leq \frac{1}{b}$.*

Definition 2.3.20 (Acyclic Logic Program) *Let P be a logic program and $\|\cdot\|$ be a level mapping such that for all ground clauses $H \leftarrow L_1 \wedge \ldots \wedge L_c \in \text{ground}(P)$ we find*

$$\|H\|_L > \|L_i\|_L.$$

Because every level mapping $\|\cdot\|$ induces a corresponding metric $m_{\mathcal{L},b}$, we use the terms *acyclic wrt.* $\|\cdot\|$ and *acyclic wrt.* $m_{\mathcal{L},b}$ as synonyms.

Lemma 2.3.21 *Let P be acyclic with respect to some level mapping $\|\cdot\|$. Then, for each level n and two interpretations I and J, we find that whenever $d(I, J) \leq 2^{-n}$ holds, $d(T_P(I), T_P(J)) \leq 2^{-(n+1)}$ follows.*

Corollary 2.3.22 *Let P be acyclic. Then we find T_P to be contractive on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$.*

Using Banach's contraction mapping theorem (Theorem 2.2.16), we can conclude that P has a unique model.

Corollary 2.3.23 *Let P be acyclic. Then we find that P has a unique model.*

Using the metric $m_{\mathcal{L},b}$ we can define the following notion of approximation for single step operators. We say that one operator approximates a second up to a given level, if and only if the outputs of both agree up to this level for any interpretation.

Definition 2.3.24 *Let $b > 2$, $m_{\mathcal{L},b}$, T_P and T_Q be given. We say that T_Q approximates the operator T_P to degree n wrt. the metric $m_{\mathcal{L},b}$, if and only if we find $m_{\mathcal{L},b}(T_Q(I), T_P(I)) \leq b^{-n}$ for all interpretations $I \in \mathcal{I}_{\mathcal{L}}$.*

Using this notion of approximation and the restricted operators from above, we obtain the following lemma. The restricted version of a given operator approximates the operator up to the level of restriction.

Lemma 2.3.25 *Let T_P be some consequence operator and $\lfloor T_P \rfloor_n$ be its restricted version of level n. Then, we find $d(\lfloor T_P \rfloor_n(I), T_P(I)) \leq 2^{-n}$ for all interpretations $I \in \mathcal{I}_{\mathcal{L}}$.*

Because the restricted version $\lfloor P \rfloor_n$ of a given covered program P is finite, we find that its T_P-operator $T_{\lfloor P \rfloor_n}$ coincides with the restricted operator $\lfloor T_P \rfloor_n$ of the program P. Therefore, we can conclude that $T_{\lfloor P \rfloor_n}$ approximates T_P to degree n.

Lemma 2.3.26 *Let P be a covered logic program and let $\lfloor P \rfloor_n$ for $n \geq 0$ be defined as above. Then $T_{\lfloor P \rfloor_n}$ approximates T_P to degree n.*

2.4 Binary Decision Diagrams

In this section, we dicuss *binary decision diagrams (BDDs)* briefly. BDDs are a widely used data-structure. Many efficient implementations like for example Buddy [www09a] and CUDD [www09b] are available. Here, we follow in principle the lecture notes of Henrik Reif Andersen [And99].

Definition 2.4.1 (BDD) *Let I be a finite set of indices with $0, 1 \in I$, let $\mathcal{V}$ be a set of propositional variables. A* binary decision diagram *is a quadruple $\langle 0, 1, R, N \rangle$ with $N \subseteq I \times \mathcal{V} \times I \times I$ and $R \in I$ such that $\mathrm{id} : N \to I : \langle i, v, h, l \rangle \mapsto i$ is a bijection between N and $I \setminus \{0, 1\}$.*

For $E = \{(i, h), (i, l) \mid \langle i, v, h, l \rangle \in N\}$ we find (I, E) to be a directed acyclic graph with sinks 0 and 1 such that any other node has an out-degree of 2.

Notation 2.4.2 *Let N be the set of nodes as defined above, let $\mathcal{V}$ be the underlying set of variables and let I be the set of identifiers, then we define the following functions:*

$$\begin{aligned} \mathrm{var} : N &\to \mathcal{V} & \mathrm{high} : N &\to I & \mathrm{low} : N &\to I \\ \langle i, v, h, l \rangle &\mapsto v & \langle i, v, h, l \rangle &\mapsto h & \langle i, v, h, l \rangle &\mapsto l \end{aligned}$$

Because id *is a bijection between N and $I \setminus \{0, 1\}$ we will use $\mathrm{var}(i) = \mathrm{var}(\mathrm{id}^{-1}(i))$, $\mathrm{high}(i) = \mathrm{high}(\mathrm{id}^{-1}(i))$ and $\mathrm{low}(i) = \mathrm{low}(\mathrm{id}^{-1}(i))$ for $i \in I \setminus \{0, 1\}$.*

Definition 2.4.3 (Corresponding Logic Formula) *Let $B = \langle 0, 1, R, N \rangle$ be a binary decision diagram with I being the underlying set of identifiers. Let F denote the set of propositional formulae. The logic* corresponding *logic formula $\mathrm{pf}(B) := \mathrm{pf}(R)$ with* pf *being recursively defined as follows:*

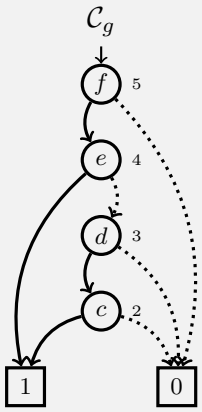
$$\mathrm{pf} : I \to F$$

$$i \mapsto \begin{cases} \bot & \text{iff } i = 0 \\ \top & \text{iff } i = 1 \\ (\mathrm{var}(i) \wedge \mathrm{pf}(\mathrm{high}(i))) \vee (\neg\, \mathrm{var}(i) \wedge \mathrm{pf}(\mathrm{low}(i))) & \text{otherwise} \end{cases}$$

Example 2.4.1 Let the BDD B be defined as $B = \langle 0, 1, R, N\rangle$ with

$$N = \{\langle 2, c, 1, 0\rangle, \langle 3, d, 2, 0\rangle, \langle 4, c, 1, 3\rangle, \langle 5, f, 4, 0\rangle\} \quad \text{and} \quad R = 5.$$

Below on the left, you find the graphical notation used throughout this thesis. All "high"-links are depicted solid, while "low"-links are shown as dotted lines. The index of the nodes is shown at the right and the associated variable within the circle. In the sequel the index of the nodes is usually omitted and to denote the root node we use a small labeled arrow instead. In this example the root node is labelled $\mathcal{C}_g$.



$$\begin{aligned}
\mathrm{pf}(2) &= (c \wedge \mathrm{pf}(1)) \vee (\neg c \wedge \mathrm{pf}(0)) \\
&= (c \wedge \top) \vee (\neg c \wedge \bot) = c \\
\mathrm{pf}(3) &= (d \wedge \mathrm{pf}(2)) \vee (\neg d \wedge \mathrm{pf}(0)) \\
&= (d \wedge c) \vee (\neg d \wedge \bot) = d \wedge c \\
\mathrm{pf}(4) &= (e \wedge \mathrm{pf}(1)) \vee (\neg e \wedge \mathrm{pf}(3)) \\
&= (e \wedge \top) \vee (\neg e \wedge (d \wedge c)) = e \vee (\neg e \wedge d \wedge c) \\
\mathrm{pf}(5) &= (f \wedge \mathrm{pf}(4)) \vee (\neg f \wedge \mathrm{pf}(0)) \\
&= (f \wedge (e \vee (\neg e \wedge d \wedge c))) \vee (\neg f \wedge \bot) \\
&= (f \wedge e) \vee (f \wedge \neg e \wedge d \wedge c) \\
\mathrm{pf}(B) &= \mathrm{pf}(5) = (f \wedge e) \vee (f \wedge \neg e \wedge d \wedge c)
\end{aligned}$$

BDDs are graphical representations of formulae. In the sequel we use a graphical notation for BDDs as shown in Example 2.4.1. To evaluate a given interpretation J, we can follow the path from the root node R. Starting with R, we follow the high-branch of the node if the corresponding variable is mapped to true under J and the low-branch otherwise. The value of the whole formula is true if this path ends at the sink 1 and false otherwise.

Definition 2.4.4 (Path) *Let $\langle 0, 1, R, N\rangle$ be a binary decision diagram. A sequence $[i_1, \ldots, i_l]$ of node-indices is called a path of length l, if $i_1 = R$, $i_l \in \{0, 1\}$ and for all i_j with $1 \leq j < l$ we find $\mathrm{high}(i_j) = i_{j+1}$ or $\mathrm{low}(i_j) = i_{j+1}$.*

Example 2.4.2 Reconsidering the BDD from Example 2.4.1, we find the following paths through the diagram and the corresponding value of the underlying formula with respect to the given interpretation:

Interpretation	Path	Value
$\{c, d, e, f\}$	$[5, 4, 1]$	$\top$
$\{c, d, e\}$	$[5, 0]$	$\bot$
$\{c, d, f\}$	$[5, 4, 3, 2, 1]$	$\top$
$\{c, d\}$	$[5, 0]$	$\bot$
$\{c, e, f\}$	$[5, 4, 1]$	$\top$
$\{c, d, e\}$	$[5, 0]$	$\bot$
$\{c, d, f\}$	$[5, 4, 3, 2, 1]$	$\top$
$\{c\}$	$[5, 0]$	$\bot$

Interpretation	Path	Value
$\{d, e, f\}$	$[5, 4, 1]$	$\top$
$\{d, e\}$	$[5, 0]$	$\bot$
$\{d, f\}$	$[5, 4, 3, 2, 0]$	$\top$
$\{d\}$	$[5, 0]$	$\bot$
$\{e, f\}$	$[5, 4, 1]$	$\top$
$\{d, e\}$	$[5, 0]$	$\bot$
$\{d, f\}$	$[5, 4, 3, 2, 0]$	$\top$
$\{\}$	$[5, 0]$	$\bot$

We introduce further restrictions on BDDs. First, we enforce the variables to be linearly ordered and the BDD to be such that the high and low node are bigger with respect to this order. In this case, we call it *ordered binary decision diagram*.

Definition 2.4.5 (OBDD) *Let $\langle 0, 1, R, N\rangle$ be a BDD, let $\mathcal{V}$ be the underlying set of variables and let $\prec$ be a linear order on V. We call $\langle \prec, 0, 1, R, N\rangle$* ordered binary decision diagram (OBDD)*, iff for all $n \in N$ the following holds: if* $\text{high}(n) \notin \{0, 1\}$ *then* $\text{var}(n) \prec \text{var}(\text{high}(n))$ *and if* $\text{low}(n) \notin \{0, 1\}$ *then* $\text{var}(n) \prec \text{var}(\text{low}(n))$.

Throughout this thesis, we use a simple Prolog implementation to construct and manipulate BDDs. A BDD is represented in a quadruple `bdd(VO, BDD0, BDD1, Nodes)`, containing the variable order (`VO`, represented as a list), the ids of the low (`BDD0`) and high (`BDD1`) sink, and the list of all nodes (`Nodes`). The predicates to construct an empty BDD (`bddEmpty/4`), to access low (`bddNode0/2`) and high sink (`bddNode1/2`), all internal nodes (`bddNode/2`) and the order of a given variable (`bddVarOrder/3`) are shown in Figure 2.4.

```
1  bddEmpty(VO, BDD0, BDD1, bdd(VO, BDD0, BDD1, [])).
2  bddNode0(bdd(_, BDD0, _, _), BDD0).
3  bddNode1(bdd(_, _, BDD1, _), BDD1).
4  bddNode(bdd(_, _, _, Nodes), Node) :- member(Node, Nodes).
5  bddVarOrder(bdd(VO, _, _, _), V, O) :- nth1(O, VO, V).
6
7  bddNodeOrder(BDD, Node, O) :-
8          bddNode(BDD, Node),
9          Node = node(V, _, _, _),
10         bddVarOrder(BDD, V, O).
```

Figure 2.4: *Implementation to construct an empty BDD and to access the ingredients of a BDD*

To enforce the construction of ordered BDDs, they are usually created recursively using the `bddApply` predicate shown in Figure 2.5 and 2.6. I.e., a new BDD is constructed from two nodes and a binary operation. The base cases, i.e., those cases where one of the two BDDs is a sink, are shown in Figure 2.6. The other cases are shown in Figure 2.5. Depending on the variable order of the root nodes of the given BDDs, one of the three cases applies, and the problem is devided in two sub-problems and a call to `bddMakeNode` as described below. All further details of those calls can be found in [And99]. To construct a BDD using an if-then-else construction we use the `bddApply/7`-predicate shown in Figure 2.7.

Further restrictions we may put on a BDD are the following minimality constraints: No two nodes should behave equal, i.e., testing the same variable and linking to the same nodes. And no redundant tests should be done, i.e., there should be no nodes with high and low branch pointing to the same node. If a BDD obeys both conditions, we call it *reduced*.

Definition 2.4.6 (ROBDD) *We call an OBDD $\langle \prec, 0, 1, R, N\rangle$* reduced ordered binary decision diagram (ROBDD)*, if* $\text{var}(u) = \text{var}(v)$, $\text{high}(u) = \text{high}(v)$ *and* $\text{low}(u) = \text{low}(v)$ *implies* $u = v$, *and* $\text{high}(u) \neq \text{low}(u)$ *holds for all* $u, v \in N$.

Reduced ordered binary decision diagrams provide unique representations for any propositional formulae defined over the underlying (ordered) set of variables. This seems to solve the

```
bddApply(BDDa, OP, N1, N2, N, BDDd) :-
        bddNodeOrder(BDDa, node(V, H1, L1, N1), O),
        bddNodeOrder(BDDa, node(V, H2, L2, N2), O),
        bddApply(BDDa, OP, L1, L2, L, BDDb),
        bddApply(BDDb, OP, H1, H2, H, BDDc),
        bddMakeNode(BDDc, node(V, H, L, N), N, BDDd).

bddApply(BDDa, OP, N1, N2, N, BDDd) :-
        bddNodeOrder(BDDa, node(V1, H1, L1, N1), O1),
        bddNodeOrder(BDDa, node(_V2, _H2, _L2, N2), O2),
        O1 < O2,
        bddApply(BDDa, OP, L1, N2, L, BDDb),
        bddApply(BDDb, OP, H1, N2, H, BDDc),
        bddMakeNode(BDDc, node(V1, H, L, N), N, BDDd).

bddApply(BDDa, OP, N1, N2, N, BDDd) :-
        bddNodeOrder(BDDa, node(_V1, _H1, _L1, N1), O1),
        bddNodeOrder(BDDa, node(V2, H2, L2, N2), O2),
        O1 > O2,
        bddApply(BDDa, OP, N1, L2, L, BDDb),
        bddApply(BDDb, OP, N1, H2, H, BDDc),
        bddMakeNode(BDDc, node(V2, H, L, N), N, BDDd).
```

Figure 2.5: *Implementation to construct a BDD* `BDDd` *from two BDDs (referenced by their root node id's* `N1` *and* `N2`*) and a given binary operation* `OP`

```
% BDD1 is the absorbing element wrt.  "or"
bddApply(BDD, or, Node1, _Node2, Node1, BDD) :- bddNode1(BDD, Node1).
bddApply(BDD, or, _Node1, Node2, Node2, BDD) :- bddNode1(BDD, Node2).
% BDD0 is the neutral element wrt.  "or"
bddApply(BDD, or, Node1, Node2, Node2, BDD) :- bddNode0(BDD, Node1).
bddApply(BDD, or, Node1, Node2, Node1, BDD) :- bddNode0(BDD, Node2).

% BDD0 is the absorbing element wrt.  "or"
bddApply(BDD, and, Node1, _Node2, Node1, BDD) :- bddNode0(BDD, Node1).
bddApply(BDD, and, _Node1, Node2, Node2, BDD) :- bddNode0(BDD, Node2).
% BDD1 is the neutral element wrt.  "or"
bddApply(BDD, and, Node1, Node2, Node2, BDD) :- bddNode1(BDD, Node1).
bddApply(BDD, and, Node1, Node2, Node1, BDD) :- bddNode1(BDD, Node2).
```

Figure 2.6: *Base cases for the construction of a BDD from two BDDs using disjunction and conjunction*

```
bddApply(BDDa, ite, V, High, Low, Node, BDDf) :-
        bddNode0(BDDa, BDD0),
        bddNode1(BDDa, BDD1),
        bddMakeNode(BDDa, node(V, BDD1, BDD0, VP), VP, BDDb),
        bddMakeNode(BDDb, node(V, BDD0, BDD1, VN), VN, BDDc),
        bddApply(BDDc, and, VP, High, H, BDDd),
        bddApply(BDDd, and, VN, Low, L, BDDe),
        bddApply(BDDe, or, H, L, Node, BDDf).
```

Figure 2.7: *The construction of a BDD from a given variable and two BDDs using the if-then-else construct*

problem of creating minimal representation for a given formula. But it turns out that the size of the ROBDD depends directly on the underlying variable order, and finding a minimal order turns out to be NP complete [BW96].

Please note that we can keep a BDD reduced while constructing it using the `bddMakeNode/4`-predicate shown in Figure 2.8.

```
1  bddMakeNode(BDD, node(_, H, H, _), H, BDD) :- !.
2  bddMakeNode(BDD, node(V, H, L, _), N, BDD) :-
3          bddNode(BDD, node(V, H, L, N)).
4  bddMakeNode(BDD, node(V, H, L, N), N, BDD2) :-
5          BDD = bdd(VO, BDD0, BDD1, Nodes),
6          newNodeId(BDD, N),
7          BDD2 = bdd(VO, BDD0, BDD1, [node(V, H, L, N)|Nodes]).
8
9  newNodeId(BDD, N) :- var(N), !,
10         newNodeId(BDD, 2, N).
11 newNodeId(BDD, N) :-
12         \+ bddNode(BDD, node(_,_,_,N)).
13
14 newNodeId(BDD, N, N2) :-
15         bddNode(BDD, node(_,_,_,N)), !,
16         N1 is N + 1,
17         newNodeId(BDD, N1, N2).
18 newNodeId(_BDD, N, N).
```

Figure 2.8: *Implementation to create of nodes while constructing a reduced BDD: Given a* `BDD` *and a node* `node(V,H,L,N)`*, a call to* `bddMakeNode/4` *will unify argument 3 and 4 with the real ID for the given node and the resulting BDD. In case* `H` *and* `L` *are equal, the resulting ID is* `H`*. If there exists a node with the given variable* `V`*, high-branch* `H` *and low-branch* `L` *then the corresponding ID* `N` *is returned. In those two cases, the given BDD is not altered. Otherwise, a new node ID is generated or the given one maintained and the node is inserted into the given BDD.*

Figure 2.4 to 2.8 show the full implementation for the construction and manipulation of reduced ordered BDDs used in this thesis. In particular we use it in Chapter 6, where a novel approach for the extraction of rules from neural networks is presented.

2.5 Connectionist Systems

Within this section, we recall some basic notions from *connectionist systems*, also called *artificial neural networks*. Connectionist systems are known for the universal approximation capabilities and for their trainability from raw data. Here, we concentrate on *feed-forward neural networks*, *radial basis function networks* and *vector-based networks*. We introduce a formal notation and discuss some basic properties.

As before, this section is not meant as a self-contained introduction to neural networks, but rather to introduce all required notations. We do not cover training methods for the introduced networks, but describe the architecture and the associated network functions only. Further details on neural networks, can be found for example in [Roj96] and [Bis95].

Each connectionist system consists of a set of units and connections between them. Activation values are propagated along those weighted connections. A unit accumulates all incoming activations, performs a simple function on this input and outputs it activation value, which is further passed through the network. The activation values of some units can be set from

outside, i.e., they serve as inputs to the network. The activation values of some units can be observed from outside. Those units form the output of the system. We treat both subsets, i.e., the input and output units, as ordered sets, this allows to feed real vectors into the network and to read real vectors from the network.

The following definition introduces artificial neural networks in a slightly more general fashion than usual in the literature. We allow each unit to have different input and activation functions.

Definition 2.5.1 (Connectionist System) *A* connectionist system *(*artificial neural network*)* $\mathcal{N}$ *is an 8-tupel* $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *with:*

$\mathcal{U}$	*being an ordered set of units*
$\mathcal{U}_{\mathsf{inp}} \subseteq \mathcal{U}$	*being a designated ordered subset of input units*
$\mathcal{U}_{\mathsf{out}} \subseteq \mathcal{U}$	*being a designated ordered subset of output units*
$\mathcal{C} \subseteq \mathcal{U} \times \mathcal{U}$	*being a set of connections between units*
$\omega : \mathcal{C} \to \mathbb{R}$	*being a mapping from connections to weights*
$\theta : \mathcal{U} \to \mathbb{R}$	*being a mapping from units to their bias*
$F^I : \mathcal{U} \to (\mathbb{R}^{\lvert\mathcal{U}\rvert} \to \mathbb{R})$	*being a mapping from units to their input function*
$F^A : \mathcal{U} \to (\mathbb{R} \times \mathbb{R} \to \mathbb{R})$	*being a mapping from units to their activation function*

Notation 2.5.2 *To simplify the notation, we use* θ_u *to denote* $\theta(u)$*,* F^A_u *for* $F^A(u)$*,* F^I_u *for* $F^I(u)$ *and* $\omega_{u \blacktriangleright v}$ *for* $\omega(c)$ *with* $c = (u, v)$*.*

This very generic definition allows to cover all usual architectures, like feed-forward networks, RBF networks and vector-based networks, as introduced in the following sections. But before proceeding to those different types of networks, we discuss a number of well known input and activation functions.

2.5.1 Input- and Activation Functions

Two important input functions, the *weighted sum* and the *distance* functions, are introduced in Definition 2.5.3 and 2.5.4 below. The first is used to compute the weighted sum of all predecessor activations, i.e., the activation values of all units are weighted (multiplied) by the weights to the unit and then added.

Definition 2.5.3 (Weighted Sum Input Function) *Let* $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *be a given network with* $u \in \mathcal{U}$ *and* $n = |\mathcal{U}|$*. We define the* weighted sum input function wsum_u *as follows:*

$$\mathrm{wsum}_u : \mathbb{R}^n \to \mathbb{R} : (x_1, \ldots, x_n) \mapsto \sum_{(v,u) \in \mathcal{C}} \omega_{v \blacktriangleright u} \cdot x_i \text{ for } v \in_i \mathcal{U}$$

The distance input function as defined below is used to compute the distance between the activations of the predecessor units and the weights to those units. For this we employ a metric m on the real vectors.

Definition 2.5.4 (Distance Input Function) *Let* $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *be a given network with* $u \in \mathcal{U}$ *and* $n = |\mathcal{U}|$ *and let* $m : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ *be a family of metrics on* $\mathbb{R}^d$ *for all* $d > 0$*. We define the* distance input function $\mathrm{dist}(m)_u : \mathbb{R}^{\mathcal{U}} \to \mathbb{R}$ *as follows:*

$$\begin{aligned} \mathrm{dist}(m)_u(x_1, \ldots, x_n) = & m(y_1, \ldots, y_d, w_1, \ldots, w_d) \\ & \text{with } (y_i, w_i) \in_i \{(x_j, \omega_{v \blacktriangleright u}) \mid (v, u) \in \mathcal{C} \text{ and } v \in_j \mathcal{U}\} \end{aligned}$$

Please note, that activation functions as introduced in Definition 2.5.1, map pairs of real numbers to real numbers. We pass the units input and its threshold as parameters into the activation function. Usually, the threshold is already incorporated while computing the input. But doing so, we allow our definitions to cover all architectures needed throughout this thesis, in particular to cover also radial basis function networks, in which the threshold has a quite different meaning than in sigmoidal networks. We discuss all details and how to use the usual activation functions described now within this framework below.

A continuous, monotone increasing and bounded function is called *squashing function.* Those functions are used as activation functions in so-called *feed-forward networks.*

Definition 2.5.5 (Squashing Function) *A function $f : \mathbb{R} \to \mathbb{R}$ is called* squashing function *if and only if it is non-constant, continuous, monotone increasing and bounded.*

Next, we introduce three often used squashing functions. Their plots are shown in Example 2.5.1.

Definition 2.5.6 (Threshold and Step Function) *For $a, b, p \in \mathbb{R}$ we define the step function $\Theta^b_{a,p}$ and the threshold function $\Theta^b_a : \mathbb{R} \to \mathbb{R}$ as follows:*

$$\Theta^b_{a,p}(x) = \begin{cases} a & \text{iff } x < p \\ b & \text{otherwise} \end{cases} \qquad \Theta^b_a(x) = \Theta^b_{a,0}(x)$$

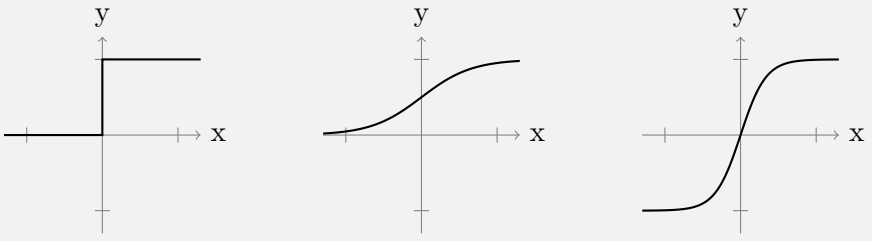
Definition 2.5.7 (Sigmoidal Function) *For $p, h, s \in \mathbb{R}$ we define the parametrised sigmoidal function* $\mathrm{sigm}^h_{p,s} : \mathbb{R} \to \mathbb{R}$ *as follows:*

$$\mathrm{sigm}^h_{p,s}(x) := \frac{h}{1 + e^{-s(x-p)}}$$

Definition 2.5.8 (Bipolar Sigmoidal Function) *The bipolar sigmoidal function (also called hyperbolic tangent)* $\tanh : \mathbb{R} \to \mathbb{R}$ *is defined as follows:*

$$\tanh(x) := \frac{2}{1 + e^{-2 \cdot x}} - 1$$

Example 2.5.1 (Squashing Functions) The plots of threshold, sigmoidal and tanh functions (left to right) as introduced in Definition 2.5.6, 2.5.7 and 2.5.8 respectively are shown below.



So-called *radial basis functions* are another class of widely used activation function. Their outputs $y = f_c(x)$ depend only on the input's distance to some point c, called the centre of f.

Definition 2.5.9 (Radial Basis Function) *Let m be a metric on $\mathbb{R}$ as introduced in Definition 2.2.1 and let $c \in \mathbb{R}$ be fixed. A function $f_c : \mathbb{R} \to \mathbb{R}$ is called a* radial basis function*, if and only if there is a function f' with $f_c(x) = f'(m(x, c))$ for all $x \in \mathbb{R}$.*

The following definitions introduce three examples of radial basis functions. Their plots are shown in Example 2.5.2.

Definition 2.5.10 (Triangular Function) *For $h, p \in \mathbb{R}$ and $w > 0 \in \mathbb{R}$, we define the* triangular function *as follows:*

$$\mathrm{tri}^{h}_{p,w} : \mathbb{R} \to \mathbb{R} : x \mapsto \begin{cases} -\frac{h}{w} \cdot |x - p| + h & \text{if } |x - p| \leq w \\ 0 & \text{otherwise} \end{cases}$$

Definition 2.5.11 (Gaussian Function) *Let $\mu \in \mathbb{R}$ and $\sigma > 0 \in \mathbb{R}$. Then, we define the* normalised Gaussian function *as follows:*

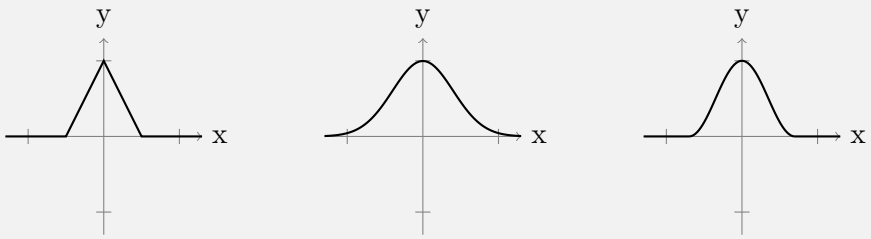
$$f_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Definition 2.5.12 (Raised Cosine Function) *For $h, p \in \mathbb{R}$ and $w > 0 \in \mathbb{R}$, we define the parametrised* raised cosine function *as follows:*

$$\mathrm{rcos}^{h}_{p,w} : \mathbb{R} \to \mathbb{R} : x \mapsto \begin{cases} \frac{h}{2} \cdot \left(1 + \cos\left(\pi \cdot \frac{x-p}{w}\right)\right) & \text{if } |x - p| \leq w \\ 0 & \text{otherwise} \end{cases}$$

Please note, that the centres of the triangular and raised cosine functions are specified by p and by μ for the Gaussian. For all three functions, we find the output to be dependent only on the distance between the input x and the respective centres. This can easily be shown formally, or by looking at the plots as shown in Example 2.5.2.

Example 2.5.2 (Radial Basis Functions) The triangular, Gaussian and raised cosine function as introduced in Definition 2.5.10, 2.5.11 and 2.5.12 respectively.



2.5.2 Feed-Forward Neural Networks

A *feed-forward neural network (FFN)* is a connectionist system in which the units are connected such that they form an acyclic directed graph. I.e., if we propagate an input vector through the network, we find that the result does not depend on previous inputs.

Definition 2.5.13 (Feed-Forward Neural Network, FFN) *Let $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ be an artificial neural network such that $\langle \mathcal{U}, \mathcal{C} \rangle$ is an acyclic directed graph. Then, $\mathcal{N}$ is called a* feed-forward neural network (FFN).

We often discuss networks applying the threshold function $\Theta^{t^+}_{t^-}$. Therefore, we introduce the following notation, to refer to those networks:

Notation 2.5.14 (Threshold Network) *Let* $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *be an artificial neural network such that*

$$F_u^I = \mathrm{wsum}_u \qquad \text{and} \qquad F_u^A(x,t) = \Theta_{t^-}^{t^+}(x+t) \qquad \text{for all } u \in \mathcal{U},$$

then we call $\mathcal{N}$ *a* threshold network *and use* $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ *to refer to it.*

Under the condition that $\langle \mathcal{U}, \mathcal{C} \rangle$ is an acyclic directed graph, we can define the input-output function of a feed-forward network as follows:

Definition 2.5.15 (Feed-Forward Network Function) *Let* $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *be a feed-forward neural network with* $|\mathcal{U}_{\mathsf{inp}}| = n$ *and* $|\mathcal{U}_{\mathsf{out}}| = m$*. The associated network function* $f_{\mathcal{N}} : \mathbb{R}^n \to \mathbb{R}^m$ *and the output function* $f_o : \mathcal{U} \times \mathbb{R}^n \to \mathbb{R}^m$ *are defined as follows:*

$$f_{\mathcal{N}} : \mathbb{R}^n \to \mathbb{R}^m : (x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_m) \quad \text{with } y_i = f_o(u, x_1, \ldots, x_n) \text{ for } u \in_i \mathcal{U}_{\mathsf{out}} \text{ and}$$

$$f_o : \mathcal{U} \times \mathbb{R}^n \to \mathbb{R} : (u, x_1, \ldots, x_n) \mapsto \begin{cases} x_j & \text{if } u \in_j \mathcal{U}_{\mathsf{inp}} \\ F_u^A(f_i(u, x_1, \ldots, x_n), \theta_u) & \text{otherwise} \end{cases}$$

$$f_i : \mathcal{U} \times \mathbb{R}^n \to \mathbb{R} : (u, x_1, \ldots, x_n) \mapsto F_u^I(y_i, \ldots, y_d) \text{ with } y_i = f_o(u_i, x_1, \ldots, x_n) \text{ for } u_i \in_i \mathcal{U}$$

Please note that $f_{\mathcal{N}}$ as introduced above is well defined, because a units input and output do only depend on the state of the predecessor units. Taking into account that the network does not contain cycles, we find that the activation value of the output units does depend on the input to the network only. I.e., on the values used to activate the input units. We continue by introducing a number of well known feed-forward network architectures. These are *sigmoidal networks*, *radial basis function networks* and *vector-based networks.*

2.5.3 3-Layer Perceptrons

Now we introduce a special case of feed-forward networks, namely so-called 3-layer perceptrons. They form the most common type of network, therefore they are sometimes just called called *feed-forward neural networks.*

Definition 2.5.16 (3-Layer Perceptron) *Let* $\mathcal{U}_{\mathsf{inp}}$, $\mathcal{U}_{\mathsf{hid}}$ *and* $\mathcal{U}_{\mathsf{out}}$ *be three pair-wise distinct sets of units, called* input, hidden *and* output layer*, respectively. Let* $\mathcal{U} = \mathcal{U}_{\mathsf{inp}} \cup \mathcal{U}_{\mathsf{hid}} \cup \mathcal{U}_{\mathsf{out}}$, $\mathcal{C} = \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \cup \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}}$ *and* $F_u^I = \mathrm{wsum}_u$ *for all* $u \in \mathcal{U}$*. Let* $f_{\mathsf{inp}}, f_{\mathsf{hid}}, f_{\mathsf{out}} : \mathbb{R} \to \mathbb{R}$ *be three given squashing functions and let*

$$F_u^A : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : (x,t) \mapsto \begin{cases} f_{\mathsf{inp}}(x+t) & \text{iff } u \in \mathcal{U}_{\mathsf{inp}} \\ f_{\mathsf{hid}}(x+t) & \text{iff } u \in \mathcal{U}_{\mathsf{hid}} \\ f_{\mathsf{out}}(x+t) & \text{iff } u \in \mathcal{U}_{\mathsf{out}} \end{cases}$$

Then, we call $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *a* 3-layer perceptron *for a given threshold functions* θ *and a given weight function* ω*.*

Notation 2.5.17 *To simplify the notation and to emphasise the layered structure, we use* $\mathsf{inp}_{id}(\theta)$, $\mathsf{hid}_{id}(\theta)$, $\mathsf{out}_{id}(\theta)$ *to denote the units of the input, hidden and output layer respectively and we use the following 8-tuple to denote 3-layer perceptrons:*

$$\langle \mathcal{U}_{\mathsf{inp}}, f_{\mathsf{inp}}, \omega_{\mathsf{inp} \blacktriangleright \mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, f_{\mathsf{hid}}, \omega_{\mathsf{hid} \blacktriangleright \mathsf{out}}, \mathcal{U}_{\mathsf{out}}, f_{\mathsf{out}} \rangle.$$

Notation 2.5.18 *If* $N = \langle \mathcal{U}_{\mathsf{inp}}, f_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, f_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}}, f_{\mathsf{out}} \rangle$ *is a three-layer perceptron with* $f_{\mathsf{inp}} = f_{\mathsf{hid}} = f_{\mathsf{out}} = f$ *we use the shorter notation*

$$\langle f, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}} \rangle.$$

Because every 3-layer perceptron is a feed forward network, they inherit the associated input-output function as introduced in Definition 2.5.15. But due to the layered structure it can be computed as follows: First, the input values (which are fed into the input units) are propagated to the hidden layer. There, all inputs are accumulated (by the weighted sum input function) and the associated activation function f_{hid} is applied. The activation values of the hidden layer are further propagated to the output layer, where f_{out} is applied. The resulting output values are the overall output of the network.

An important property of connectionist systems is their *universal approximation capability* as formalised in the following theorem. We restate this famous theorem by Funahashi here, because it is of central importance in Section 7.1. We restate the theorem using the notations introduced above.

Theorem 2.5.19 (Funahashi's Theorem [Fun89]) *Let* ϕ *be a squashing function as introduced in Definition 2.5.5,* $K \subset \mathbb{R}^n$ *be compact,* $f : K \to \mathbb{R}$ *be continuous and let* $\varepsilon > 0$*. Then there exists an integer* N *and real constants* c_i, θ_i *for* $1 \leq i \leq N$ *and* $w_{i,j}$ *for* $1 \leq i \leq N$ *and* $1 \leq j \leq n$ *such that*

$$\tilde{f}(x_1, \ldots, x_n) = \sum_{i=1}^{N} c_i \cdot \phi \left(\sum_{j=1}^{n} w_{i,j} \cdot x_j - \theta_i \right)$$

satisfies

$$\max_{x \in K} \left| f(x_1, \ldots, x_n) - \tilde{f}(x_1, \ldots, x_n) \right| \leq \varepsilon$$

Funahashi's theorem can also be read as follows: For any real-valued continuous function f, defined on a compact subset of $\mathbb{R}^n$ and any margin bound $\varepsilon > 0$, there exists a 3-layer perceptron $\mathcal{N} = \langle \mathcal{U}_{\mathsf{inp}}, \mathrm{id}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \phi, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}}, \mathrm{id} \rangle$ whose input and output layer apply the identity and whose hidden layer units a given squashing function ϕ such that the associated network function $f_{\mathcal{N}}$ and f agree up to ε. I.e., every such function can be approximated arbitrarily well using this type of 3-layer network.

2.5.4 Radial Basis Function Networks

A second widely used type of connectionist system, are so-called *radial basis function networks*. As already mentioned above, radial basis functions are functions whose output depends on the inputs distance to some given centre. Definitions 2.5.10 to 2.5.12 introduce three examples of such functions. To each of those functions we can associate a width, w and σ for the three examples. The threshold of the units is used as width parameter.

Definition 2.5.20 (Radial Basis Function Network, RBF Network) *Let* $\mathcal{U}_{\mathsf{inp}}$*,* $\mathcal{U}_{\mathsf{hid}}$ *and* $\mathcal{U}_{\mathsf{out}}$ *be three pair-wise distinct sets of units, called* input, hidden *and* output layer*, respectively. Let* $\mathcal{U} = \mathcal{U}_{\mathsf{inp}} \cup \mathcal{U}_{\mathsf{hid}} \cup \mathcal{U}_{\mathsf{out}}$ *and* $\mathcal{C} = \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \cup \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}}$*. Let* $m : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ *be a metric on* $\mathbb{R}^d$ *for all* $d > 0$ *and let* $f_w : \mathbb{R} \to \mathbb{R}$ *be a radial basis function with associated width parameter*

w.

$$F_u^I = \begin{cases} \text{wsum}_u & \text{iff } u \in \mathcal{U}_{\mathsf{inp}} \cup \mathcal{U}_{\mathsf{out}} \\ \text{dist}(m)_u & \text{iff } u \in \mathcal{U}_{\mathsf{hid}} \end{cases}$$

$$F_u^A : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : (x,t) \mapsto \begin{cases} x + t & \text{iff } u \in \mathcal{U}_{\mathsf{inp}} \cup \mathcal{U}_{\mathsf{out}} \\ f_t(x) & \text{iff } u \in \mathcal{U}_{\mathsf{hid}} \end{cases}$$

Then, we call $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *an* radial basis function network (RBF network) *for a given threshold functions* θ *and a given weight function* ω.

As for 3-layer perceptrons above, we introduce another notation for radial basis function networks. Please note, that the activation functions for input and output layer are fixed to be the sum of threshold and input. Furthermore, all input functions are fixed to be either the weighted sum (input and output layer) or the distance input function (hidden layer). Therefore, we use the following notation to denote RBF networks.

Notation 2.5.21 *We use* $\langle f_w, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp} \blacktriangleright \mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid} \blacktriangleright \mathsf{out}}, \mathcal{U}_{\mathsf{out}} \rangle$ *to denote the RBF network corresponding to* $\mathcal{U}_{\mathsf{inp}}$, $\mathcal{U}_{\mathsf{hid}}$, $\mathcal{U}_{\mathsf{out}}$, θ, ω *and* f_w *as in Definition 2.5.20.*

Due to their layered structure, we can use the network function introduced above for RBF networks as well.

2.5.5 Vector-Based Neural Network

Finally, we discuss a third family of connectionist systems, namely so-called *vector-based networks*. An overview of this type of networks is given in [Fri98].

Similar to RBF networks as introduced above, they consist of three layers and every hidden unit has an area of influence. As before, the location of this area is determined by the weights between input and hidden layer. But in contrast to RBF networks, only one unit is allowed to contribute to the overall output of the system, namely the *winner unit*. It is the unit for which the *reference vector*, determined by its incoming weights is closest to the current input. This unit is then allowed to output a value (usually 1.0) which is propagated to the output layer. I.e., the overall output of the system is determined by the outgoing connectionist of the winner unit.

Definition 2.5.22 (Vector-based Network) *Let* $\mathcal{U}_{\mathsf{inp}}$, $\mathcal{U}_{\mathsf{hid}}$ *and* $\mathcal{U}_{\mathsf{out}}$ *be three pair-wise distinct sets of units, called* input, hidden *and* output layer, *respectively. Let* $\mathcal{U} = \mathcal{U}_{\mathsf{inp}} \cup \mathcal{U}_{\mathsf{hid}} \cup \mathcal{U}_{\mathsf{out}}$ *and* $\mathcal{C} = \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \cup \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}}$. *Let* $m : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ *be a metric on* $\mathbb{R}^d$ *for all* $d > 0$.

$$F_u^I = \begin{cases} \text{wsum}_u & \text{iff } u \in \mathcal{U}_{\mathsf{inp}} \cup \mathcal{U}_{\mathsf{out}} \\ \text{dist}(m)_u & \text{iff } u \in \mathcal{U}_{\mathsf{hid}} \end{cases}$$

$$F_u^A : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : (x,t) \mapsto \begin{cases} x & \text{iff } u \in \mathcal{U}_{\mathsf{inp}} \cup \mathcal{U}_{\mathsf{out}} \\ 1 & \text{iff } u \in \mathcal{U}_{\mathsf{hid}} \text{ and } u \text{ is winner} \\ 0 & \text{iff } u \in \mathcal{U}_{\mathsf{hid}} \text{ and } u \text{ is not winner} \end{cases}$$

$$\theta_u = 0 \text{ for all } u \in \mathcal{U}$$

Then, we call $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ *a* vector-based network *for a given weight function* ω.

Notation 2.5.23 (Reference Vector) *Let $u \in \mathcal{U}_{\mathsf{hid}}$ be a hidden unit of a given vector based network. We use the term* reference vector *to denote the vector constituted by its input weights*

$$(v_1, \ldots, v_n) \text{ with } v_i = \omega_{v \blacktriangleright u} \text{ for } v \in_i \mathcal{U}_{\mathsf{inp}}$$

As above, we introduce a special notation for vector based networks emphasising the layered structure and the facts that the threshold are not needed and all input and activation functions are fixed.

Notation 2.5.24 *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, F^I, F^A \rangle$ a vector-based network for some metric m. Then we use $\langle m, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp} \blacktriangleright \mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid} \blacktriangleright \mathsf{out}}, \mathcal{U}_{\mathsf{out}} \rangle$ as a short-hand notation for $\mathcal{N}$*

3 Embedding Propositional Rules into Connectionist Systems

... where we show how to embed propositional rules into connectionist systems. First, we review the construction of threshold networks. They are constructed such that the input-output behaviour of the network mimics the semantic operator of the underlying program. Instead of the hyperbolic tangent and the sigmoidal function we use the threshold functions Θ_{-1}^{+1} and Θ_{0}^{1}, respectively. Those functions together with their threshold versions are depicted in Figure 3.1. In Section 3.2, we show how to extend the constructions to networks with smooth activation function. Both sections start with a discussion of an appropriate embedding of interpretations into vectors of real numbers and an extraction thereof. Afterwards, the resulting networks are introduced formally and the behaviour-equivalence is shown. Finally, we discuss the iteration of the connectionist computation.

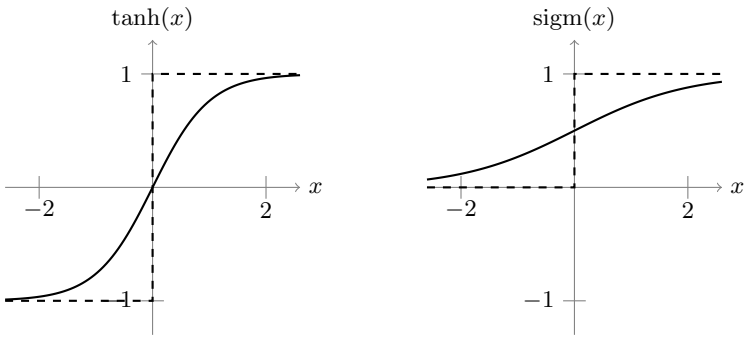


Figure 3.1: *Activation functions (*tanh *on the left and* sigm *on the right) used throughout this chapter: Their threshold counterparts (Θ_{-1}^{+1} and Θ_{0}^{1}) are depicted by dashed lines.*

3.1 Embedding Semantic Operators into Threshold Networks

Here, we show how to construct a so-called *Threshold PropCore-Network* for a given propositional logic program P, such that the program computes the associated immediate consequence operator T_P. After discussing the link between interpretations and real vectors in detail, we give a formal definition of threshold PropCore-networks and discuss their properties. Then, we present an algorithm that actually constructs such a network. This network is finally transformed into a recurrent one, allowing for the iteration of the embedded T_P-operator.

To link the space of propositional interpretations with the space of real vectors, we use the *propositional embedding* presented in Definition 3.1.1. For a finite set of propositional variables $\mathcal{V}$ with cardinality m, we associate an m-dimensional vector $\boldsymbol{\iota}(I)$ to each interpretation $I \in \mathcal{I}_\mathcal{V}$. Each variable $v \in \mathcal{V}$ is mapped to one dimension (say i) and the corresponding value $\boldsymbol{\iota}^{[i]}(I)$ is set to t^+ if $I \models v$ and to t^- if $I \not\models v$, where $t^- < t^+ \in \mathbb{R}$.

Definition 3.1.1 (Propositional Embedding) *Let $\mathcal{V}$ be a finite set of propositional variables with cardinality m and $t^- < t^+ \in \mathbb{R}$. Furthermore, let $|\cdot|$ be a fixed bijection between $\mathcal{V}$ and $\{1,\ldots,m\}$. We define the m-dimensional embedding $\boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)}(I)$ of $I \in \mathcal{I}_\mathcal{V}$ as follows:*

$$\boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)} : \mathcal{I}_\mathcal{V} \to \{t^-,t^+\}^m$$
$$I \mapsto (\boldsymbol{\iota}^{[1]}(I),\ldots,\boldsymbol{\iota}^{[m]}(I)) \qquad \text{with} \qquad \boldsymbol{\iota}^{[i]}(I) = \left\{ \begin{array}{ll} t^+ & \text{if } I \models a \\ t^- & \text{otherwise} \end{array} \right\} \text{ and } |a| = i$$

To keep the notation simple we sometimes omit the subscripts $\mathcal{V}$ and $|\cdot|$. Furthermore, we usually omit the superscript (t^-,t^+) and use $t^- = -1$ and $t^+ = +1$, unless stated otherwise. Obviously, $\boldsymbol{\iota}$ is a bijective mapping from the space of interpretations to the m-dimensional vectors over $\{t^-,t^+\}$ as stated in the following lemma.

Lemma 3.1.2 *$\boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)}$ is a bijection between $\mathcal{I}_\mathcal{V}$ and $\{t^-,t^+\}^m$.*

Proof To show that $\boldsymbol{\iota} = \boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)}$ is bijective, we have to show that it is (i) injective, i.e., from $\boldsymbol{\iota}(I) = \boldsymbol{\iota}(J)$ follows that $I = J$, and (ii) surjective, i.e., for each element $x \in \{t^-,t^+\}^m$ there is an $I \in \mathcal{I}_\mathcal{V}$ such that $\boldsymbol{\iota}(I) = x$. (i) Let us assume we have $\boldsymbol{\iota}(I) = \boldsymbol{\iota}(J)$ and $I \neq J$. From $I \neq J$, we know that there must be some $v \in \mathcal{V}$ with $|v| = i$ and $v \in I$ and $v \notin J$ (or vice versa). Then we find $\boldsymbol{\iota}^{[i]}(I) = t^+$ and $\boldsymbol{\iota}^{[i]}(J) = t^-$.

With $t^- \neq t^+$, this contradicts our assumption and we can conclude that $\boldsymbol{\iota}$ is injective. (ii) Let $x \in \{t^-, t^+\}^m$. We can construct an $I \in \mathcal{I}_\mathcal{V}$ with $\boldsymbol{\iota}(I) = x$ as follows: For each $1 \leq i \leq m$ we add v to I, if $|v| = i$ and $x^{[i]} = t^+$. Hence, we can conclude that $\boldsymbol{\iota}$ is a bijection from $\mathcal{I}_\mathcal{V}$ to $\{t^-, t^+\}^m$ with $|\mathcal{V}| = m$. □

Example 3.1.1 For $\mathcal{V} = \{a, b, c\}$ with $|a| = 1, |b| = 2, |c| = 3$ and $t^- = -1$, $t^+ = +1$, we obtain the following embeddings:

$\mathbf{I}$	$\boldsymbol{\iota}(\mathbf{I})$
$\emptyset$	$(-1,-1,-1)$
$\{a\}$	$(+1,-1,-1)$
$\{b\}$	$(-1,+1,-1)$
$\{c\}$	$(-1,-1,+1)$

$\mathbf{I}$	$\boldsymbol{\iota}(\mathbf{I})$
$\{a,b\}$	$(+1,+1,-1)$
$\{a,c\}$	$(+1,-1,+1)$
$\{b,c\}$	$(-1,+1,+1)$
$\{a,b,c\}$	$(+1,+1,+1)$

This embedding allows to feed propositional interpretations over some set $\mathcal{V}$ directly into an appropriately set up network. For each propositional variable there must be a unit in the input and output layer of the network. An interpretation I is fed into the network by setting the input unit activations accordingly. Furthermore, we can read the output of the network as another interpretation, which turns out to be $T_P(I)$. For each clause in the program, there is a hidden unit. This unit is active, whenever all conditions of the rule are met, i.e., if the body of the clause is true under the current interpretation. We use the phrase *current interpretation* to denote the interpretation corresponding to the current activation pattern of the input layer. As introduced in Section 2.3, we understand a program as an ordered list of clauses, and we use $C \in_n P$ to state that C is the n-th clause of the program P. We use $\#^a_{\text{Body}}$ and $\#^{\neg a}_{\text{Body}}$ to denote the number of positive and negative occurrences of a in Body respectively. Furthermore, we use $\#^+_{\text{Body}}$ and $\#^-_{\text{Body}}$ to denote the number of all positive and the number of all negated atoms in Body and $\#^a_P$ to denote the number of clauses with head a in the program P. The following definition introduces *Threshold PropCore-Network*.

Definition 3.1.3 (Threshold PropCore-Network) *Let P be a propositional logic program with the underlying set of atoms $\mathcal{V}$ and let $t^- < t^\circ < t^+ \in \mathbb{R}$. The corresponding* Threshold PropCore-network $\text{tpcn}^{t^-,t^\circ,t^+}_\mathcal{V}(P)$ *is defined as follows:*

$$
\begin{aligned}
\text{tpcn}^{t^-,t^\circ,t^+}_\mathcal{V}(P) &:= \left\langle \Theta^{t^+}_{t^-}, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}} \right\rangle \textit{ with} \\
\mathcal{U}_{\mathsf{inp}} &:= \{\mathsf{inp}_a(t^\circ) \mid a \in \mathcal{V}\} \\
\mathcal{U}_{\mathsf{hid}} &:= \{\mathsf{hid}_n(\theta_n) \mid (h \leftarrow \mathit{Body}) \in_n P \textit{ and } \theta_n = \#^+_{\mathit{Body}} \cdot t^+ - \#^-_{\mathit{Body}} \cdot t^- - t^+ + t^\circ\} \\
\mathcal{U}_{\mathsf{out}} &:= \{\mathsf{out}_a(\theta_a) \mid a \in \mathcal{V} \textit{ and } \theta_a = \#^a_P \cdot t^- + t^\circ - t^-\} \\
\omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}} &: \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \to \mathbb{R} \\
&\quad (\mathsf{inp}_a, \mathsf{hid}_n) \mapsto \#^a_{\mathit{Body}} - \#^{\neg a}_{\mathit{Body}} \textit{ with } (h \leftarrow \mathit{Body}) \in_n P \\
\omega_{\mathsf{hid}\blacktriangleright\mathsf{out}} &: \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}} \to \mathbb{R} \\
&\quad (\mathsf{hid}_n, \mathsf{out}_a) \mapsto \begin{cases} 1.0 & \textit{if } a \textit{ is the head of clause number } n \\ 0.0 & \textit{otherwise} \end{cases}
\end{aligned}
$$

In principle, this definition captures the ideas first presented in [HK94]. In fact, by using $t^- = 0$, $t^\circ = 0.5$ and $t^+ = 1$, we would obtain exactly the same networks.

As before, we omit the sub- and superscripts if the underlying language and the values for t^-, t° and t^+ can be determined from the context. Please note, that every PropCore-network is a fully connected 3-layer feed-forward network as introduced in Definition 2.5.16, even though many connections have weight 0. The following proposition states that the network actually computes T_P for a given program P. Two timesteps after feeding an interpretation I into the network, we can read of $T_P(I)$ from the output layer.

After showing the correspondence between a logic program and the resulting PropCore-network formally, we present a Prolog implementation. Figure 3.2 shows the definition of the `tpcn/6` predicate which implements Definition 3.1.3. Instantiating `Tn`=t^-,`T0`=t°, `Tp`=t^+ and `V` with the list of propositional variables of the program `Prog` results in a binding for `Network`, which describes the corresponding $\mathrm{tpcn}_{\mathcal{V}}^{t^-,t^\circ,t^+}(P)$-network. A sample-run is shown in Example 3.1.2. The following proposition establishes the desired relation between T_P and f_P.

Proposition 3.1.4 *Let P be a logic program, let $\mathcal{V}$ be the underlying set of atoms with $|\mathcal{V}| = m$ and let $t^- < t^\circ < t^+ \in \mathbb{R}$. Furthermore, let $\mathrm{tpcn}_{\mathcal{V}}^{t^-,t^\circ,t^+}(P)$ be the corresponding PropCore-network and let $f_P : \mathbb{R}^m \to \mathbb{R}^m$ be the function computed by that network. Then the following diagram commutes:*

$$\begin{array}{ccc} I & \xrightarrow{T_P} & T_P(I) \\ \boldsymbol{\iota}\downarrow & & \downarrow\boldsymbol{\iota} \\ x & \xrightarrow[f_P]{} & f_P(x) \end{array}$$

I.e., we find for $\boldsymbol{\iota} = \boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)}$ that $\boldsymbol{\iota}(T_P(I)) = f_P(\boldsymbol{\iota}(I))$ holds for all $I \in \mathcal{I}_{\mathcal{V}}$.

Proof This proof is split for clarity in four parts. In (i) we show that a hidden unit hid_c, constructed for the c-th clause $h \leftarrow \mathrm{Body}$, is active if and only if the input unit's activation pattern, understood as an interpretation, is a model for that body. Next (ii), we show that an output unit out_a is active if and only if at least one hidden unit corresponding to a clause with head a was active at the previous time step. From (i) and (ii) it follows (iii) that after presenting some interpretation I to the network at time t, the output unit's activation pattern at time $t+2$ corresponds to $T_P(I)$. In (iv) we finally conclude that $\boldsymbol{\iota}(T_P(I)) = f_P(\boldsymbol{\iota}(I))$ and $T_P(I) = \boldsymbol{\iota}^{-1}(f_P(\boldsymbol{\iota}(I)))$.

(i) Let hid_c be a hidden unit constructed for the clause $h \leftarrow \mathrm{Body}$. According to Definition 3.1.3, we find that the threshold of the unit is $\theta_a = \#^+_{\mathrm{Body}} \cdot t^+ - \#^-_{\mathrm{Body}} \cdot t^- - t^+ + t^\circ$. As the input units output either t^+ or t^-, we can conclude that every input unit corresponding to a positive atom within the body must be active at time t and all units corresponding to negated atoms are inactive at time t. Under the assumption that the input units activation pattern is not a model for the body we find that the unit c is inactive. There would be at least one positive atom in the body such that the corresponding input unit would be inactive, or there would be a negated atom with an active input unit. Therefore, the total input of unit hid_c would not exceed the threshold. For the case that the unit hid_c is active we can conversely conclude that all units corresponding to positive atoms within Body must be active, and all units for negated atoms must be inactive. Otherwise, the units activation would not exceed the threshold.

(ii) Let out_a be an output unit constructed for some atom a. According to Definition 3.1.3, we find that the threshold $\theta_a = n \cdot t^- + t^\circ - t^-$, where n is the number of clauses in P with head a. Furthermore, we find that the weight associated with a connection from a hidden unit constructed for those clauses is 1 and all other incoming weights

are 0. Therefore, we can conclude that the only possibility for $\mathtt{out}_a$ to receive some input $< \theta_a$ is that all hidden units corresponding to clauses with head a are inactive. This allows to conclude that $\mathtt{out}_a$ is active if and only if at least one of those hidden units is active.

(iii) After presenting some interpretation I to the network at time t, we find at time $t+1$, all those hidden units to be active that were constructed for a clause with a body which is true under I. All other hidden units are inactive. From (ii) we know that this activates all those output units $\mathtt{out}_a$ at time $t+2$, for which there is at least one clause in P with head a and a body being true under I. Therefore, we can conclude that the interpretation corresponding to the output units activation pattern at time $t+2$ coincides with $T_P(I)$.

(iv) Due to the fact that the activation of the m output units is either t^+ or t^-, we know that the codomain of f_P is $\{t^-, t^+\}^m$. Furthermore, we know that the interpretation corresponding to the output units activation pattern at time $t+2$ coincides with $T_P(I)$. Therefore, we can conclude that $\boldsymbol{\iota}(T_P(I)) = f_P(\boldsymbol{\iota}(I))$. □

From the previous proposition, the facts that $\boldsymbol{\iota}$ is bijective and that f_P is a function on $\{t^-, t^+\}$, we conclude that we can compute T_P using f_P. In fact, this is the desired dependency, because we want to obtain a connectionist model to compute T_P.

Corollary 3.1.5 *For $\boldsymbol{\iota}$ and f_P as defined above, we find $T_P(I) = \boldsymbol{\iota}^{-1}(f_P(\boldsymbol{\iota}(I)))$ for all $I \in \mathcal{I}_\mathcal{L}$.*

```
1  tpcn(Tn, To, Tp, V, Prog, Network) :-
2       Ui  := { unit(A, To) | A in V },
3       Uh  := { unit(N, Th) | clause(_,B) in Prog at N,
4                              Nn := #(not(_), B),
5                              Np := size(B) - Nn,
6                              Th := Np*Tp - Nn*Tn - Tp + To },
7       Uo  := { unit(A, Th) | A in V,
8                              Th := #(clause(A,_), Prog)*Tn + To - Tn },
9       I2H := { A/N/W       | clause(_,B) in Prog at N,
10                             A in V,
11                             W := #(A, B) - #(not(A), B) },
12      H2O := { N/A/W       | clause(H,_) in Prog at N,
13                             A in V,
14                             W := ((H = A) ? 1.0 : 0.0) },
15      Network = ffn(th(Tn,Tp), Ui, I2H, Uh, H2O, Uo).
```

Figure 3.2: *Implementation to construct a behaviour-equivalent artificial neural network for a given propositional logic program* `Prog`*, the underlying set of propositional variables* `V` *and the threshold-values (*`Tn`$=t^-$,`To`$=t^\circ$ *and* `Tp`$=t^+$*).*

The termination of the `tpcn/6` algorithm follows directly from the fact that every program is a finite set of clauses and that every clause has only finitely many literals in the body. The correctness of the algorithm could be proven by showing that the result of the algorithm, if applied to some program P, coincides with the corresponding PropCore-Network for P as given in Definition 3.1.3. Obviously, both properties depend on the correctness of Prolog, but the proof is beyond the scope of this thesis. Therefore, we state the following conjecture:

Conjecture 3.1.6 *The algorithm given in Figure 3.2 always terminates and is correct. Let P be a propositional logic program and let N be the result (i.e., the network corresponding to the value of* `Network`*). Then, N computes the T_P-operator associated with P.*

The following corollary follows directly from our conjecture. Knowing that the algorithm from Figure 3.2 terminates and is correct, we can conclude that there is a 3-layer feed-forward network N for every program P, such that N computes T_P. This result was first obtained as Proposition 4 in [HK94] (see also Theorem 3.2 in [HHS04]). The result here is slightly more general as arbitrary threshold functions are allowed.

Corollary 3.1.7 *For each propositional logic program P there exists a 3-layer feed-forward network with threshold units which computes T_P and can be constructed.*

The networks constructed above, apply threshold functions within their units, which simplified the proofs, but which prevents the application of standard gradient based learning algorithms. Therefore, we extend the approach to smooth activation functions in the following section.

As already mentioned above, the constructions are in principle those presented in [HK94]. But we did not fix the parameters of the threshold function. This allows to extend those results in a straight forward fashion to networks with different smooth activation functions.

3.2 Embedding Semantic Operators into Sigmoidal Networks

Now we show how to construct networks with smooth activation functions. To carry over the results of the previous section, we would like to have a network with a similar behaviour. I.e., a hidden unit hid_n is active, if and only if the body of the n-th clause in the program is true under the current interpretation. Furthermore, an output unit out_a is active, if and only if at least one hidden unit corresponding to a clause with head a is active. For this, we need new notions of *activity* and *inactivity*. These can be defined using the thresholds $a^- < a^+ \in \mathbb{R}$, respectively. A unit is considered active, if its activation value is above or equal to a^+ and to be inactive, if it is below or equal to a^-. Whenever the activation of a unit is within (a^-, a^+), we call it *undecided*. For technical reasons, we need another state, called $\circ$*-state*, for the activation value a° with $a^- < a^\circ < a^+$.

The constructions presented below are closely related to those of Artur d'Avila Garcez and Gerson Zaverucha presented in [dGZ99]. But we use a^+ and a^- as parameters, instead of computing them. This allows more flexibility and yields new insights into the dependencies.

Definition 3.2.1 *Let u be a unit with (squashing) activation function $s : \mathbb{R} \to \mathbb{R}$, let $s^- := \lim_{x\to-\infty} s(x)$ and $s^+ := \lim_{x\to\infty} s(x)$. Let act_u be the activation value of u. Let a^-, a° and a^+ be real numbers with $s^- < a^- < a^\circ < a^+ < s^+$. We call u* active *if and only if we find $\mathrm{act}_u \geq a^+$,* inactive *for $\mathrm{act}_u \leq a^-$, to be in the $\circ$-state for $\mathrm{act}_u = a^\circ$ and to be* undecided *for $a^- < (\mathrm{act}_u \neq a^\circ) < a^+$. Furthermore, we call $i^+ := s^{-1}(a^+)$, $i^- := s^{-1}(a^-)$ and $i^\circ := s^{-1}(a^\circ)$ the* minimal activation input*,* maximal inactivation input *and* $\circ$-state input*, respectively.*

The following lemma links the inputs and activation states introduced in Definition 3.2.1 in the expected and straight forward way.

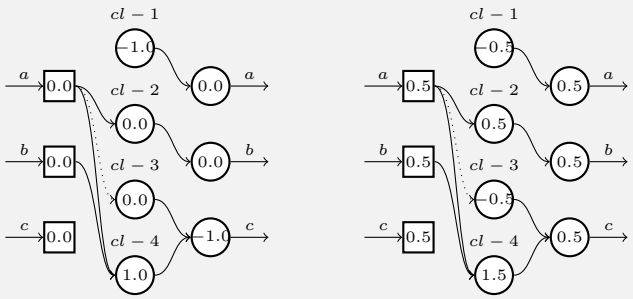
Lemma 3.2.2 *Let u, i^+, i°, i^- be as in Definition 3.2.1 and let inp_u be the input value of u. We find u to be active iff $\mathrm{inp}_u \geq i^+$, to be inactive iff $\mathrm{inp}_u \leq i^-$ and to be in the $\circ$-state iff $\mathrm{inp}_u = i^\circ$.*

Example 3.1.2 The execution of `tpcn/6` for our running example is shown below:

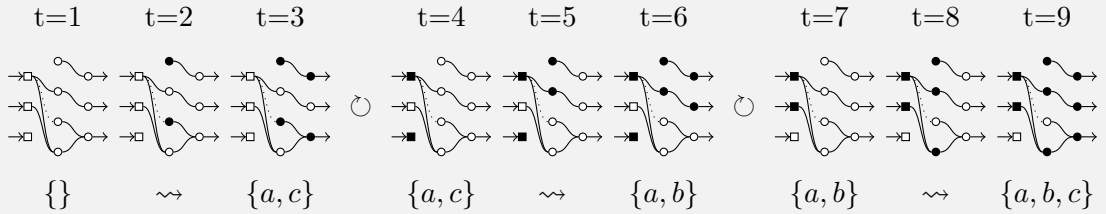
```
?- P = [clause(a,[]), clause(b,[a]), clause(c,[not(a)]), clause(c,[a,b])],
   tpcn(-1, 0, +1, [a,b,c], P, Network).

Network = ffn(th(-1,+1),
  % input layer units
  [ unit(a, 0), unit(b, 0), unit(c, 0) ],
  % connections from input to hidden layer:  Source/Target/Weight
  [ a/1/0, a/2/1, a/3/-1, a/4/1, b/1/0, b/2/0, b/3/0,  b/4/1,
    c/1/0, c/2/0, c/3/0,  c/4/0)],
  % hidden layer units
  [ unit(1, -1), unit(2, 0), unit(3, 0), unit(4, 1) ],
  % connections from hidden to output layer
  [ 1/a/1, 1/b/0, 1/c/0, 2/a/0, 2/b/1, 2/c/0,
    3/a/0, 3/b/0, 3/c/1, 4/a/0, 4/b/0, 4/c/1)],
  % output layer units
  [ unit(a, 0), unit(b, 0), unit(c, -1) ]
)
```

For convenience we give the graphical notation for the resulting network for Θ_{-1}^{+1} in the middle and for Θ_0^1 on the right:

$(cl-1)$ $a.$
$(cl-2)$ $b \leftarrow a.$
$(cl-3)$ $c \leftarrow \neg a.$
$(cl-4)$ $c \leftarrow a \wedge b.$

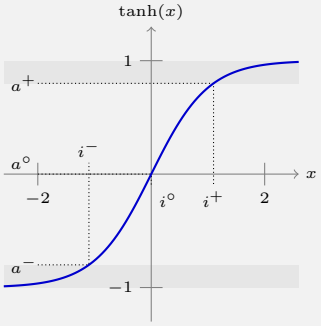


Only connections with weight 1 (solid) and weight -1 (dotted) are shown. All other connections (i.e., those with weight 0) are left out. Below, three computations of the T_P-operator are shown. At time $t = 1$, an empty interpretation is fed into the network. This causes two hidden units to become active (at $t = 2$), corresponding to clause number 1 and 3. At time $t = 3$, we can read of $T_P(\emptyset) = \{a, c\}$ from the output layer. This interpretation is fed back into the input layer at $t = 4$, restarting the computation. Active units are depicted black and inactive ones are left blank.



Example 3.2.1 Some activation and input values and the resulting plot for the tanh:

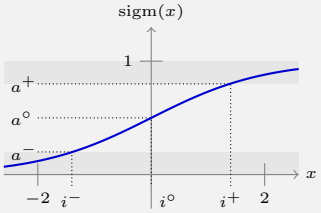
tanh	
$s^+ = 1.0$	
$a^+ = 0.8$	$i^+ \approx 1.1$
$a^\circ = 0.0$	$i^\circ = 0.0$
$a^- = -0.8$	$i^- \approx -1.1$
$s^- = -1.0$	



The upper and lower gray rectangles mark the active and inactive regions, respectively.

Example 3.2.2 Some activation and input values together with the resulting plot for the sigmoidal activation function:

sigm	
$s^+ = 1.0$	
$a^+ = 0.8$	$i^+ \approx 1.4$
$a^\circ = 0.0$	$i^\circ = 0.5$
$a^- = 0.2$	$i^- \approx -1.4$
$s^- = 0.0$	



Proof For some $\mathrm{inp}_u \geq i^+$ we find $\mathrm{act}_u = s(\mathrm{inp}_u) \geq a^+$ and hence the unit u to be active. Analogously, we find a unit to be inactive for $\mathrm{inp}_u \leq i^-$ and to be in the $\circ$-state of $\mathrm{inp}_u = i^\circ$. □

Using the activation ranges just introduced, we can define a new inverse embedding function. Instead of mapping vectors over $\{t^-, t^+\}$ to interpretations (as done by $\boldsymbol{\iota}^{-1}$ in Lemma 3.1.2), we map vectors of real numbers to interpretations. The resulting interpretation contains all those atoms whose corresponding elements in the vector are greater or equal to a^+. If a vector contains elements $a^- < x < a^+$, the value $\bowtie$ is returned, indicating that there is no corresponding interpretation.

Definition 3.2.3 *Let $\mathcal{V}$ be a finite set of propositional variables with $|\mathcal{V}| = m$. As in Definition 3.1.1, we assume $|\cdot|$ to be a bijection between $\mathcal{V}$ and $\{1, \ldots, m\}$. The inverse embedding function $\kappa_{|\cdot|}^{(a^-,a^+)}$ is defined as follows:*

$$\kappa_{|\cdot|}^{(a^-,a^+)} : \mathbb{R}^m \to \mathcal{I}_{\mathcal{V}} \cup \{\bowtie\}$$

$$(x^{[1]}, \ldots, x^{[m]}) \mapsto \begin{cases} \bowtie & \text{if there is some } a^- < x^{[i]} < a^+ \\ \{v \mid v \in \mathcal{V} \text{ and } x^{[|v|]} \geq a^+\} & \text{otherwise} \end{cases}$$

Example 3.2.3 For $\mathcal{V} = \{a, b, c\}$ with $|a| = 1, |b| = 2, |c| = 3$ and $\mathbf{x} = (-0.9, 0.9, 0.6)$ we find

$$\kappa_{|\cdot|}^{(-0.5,0.5)}(\mathbf{x}) = \{b, c\} \qquad \text{and} \qquad \kappa_{|\cdot|}^{(-0.8,0.8)}(\mathbf{x}) = \bowtie .$$

Obviously, $\kappa_{|\cdot|}^{(a^-,a^+)}$ and $\boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)}$ are compatible for $t^- \leq a^- < a^+ \leq t^+ \in \mathbb{R}$ as stated in the following lemma. For an embedded interpretation $\mathbf{x} = \boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)}(I)$ we can extract the same interpretation using $\kappa_{|\cdot|}^{(a^-,a^+)}$ and find $\kappa_{|\cdot|}^{(a^-,a^+)}(\mathbf{x}) = I$.

Lemma 3.2.4 *For all $I \in \mathcal{I}_\mathcal{V}$ and all $t^- \leq a^- < a^+ \leq t^+ \in \mathbb{R}$, we find*

$$\kappa_{|\cdot|}^{(a^-,a^+)}\left(\boldsymbol{\iota}_{|\cdot|}^{(t^-,t^+)}(I)\right) = I.$$

Proof From $t^- \leq a^- < a^+ \leq t^+$, we can conclude that $\kappa(\boldsymbol{\iota}(I)) \neq \bowtie$, i.e., that there is a corresponding interpretation for $\boldsymbol{\iota}(I)$. From $a \in \kappa(\boldsymbol{\iota}(I))$ with $|a| = i$, we know that $x^{[i]} \geq a^+ > t^-$ and hence $a \in I$ and $\kappa(\boldsymbol{\iota}(I)) \subseteq I$. Conversely, from $a \in I$ and $|a| = i$, we get $x^{[i]} = t^+ \geq a^+$ and hence $a \in \kappa(\boldsymbol{\iota}(I))$ and $I \subseteq \kappa(\boldsymbol{\iota}(I))$. Alltogether, we obtain $\kappa(\boldsymbol{\iota}(I)) = I$. □

Unfortunately, the construction presented in the previous section does not guarantee that a unit is either active or inactive. Assuming a^+ to be 0.4, we find that unit *cl*–4 in Example 3.1.2 in the middle to be inactive, even if unit a and b are both active. But as shown below, it is possible to modify the weights such that each unit is either active, or inactive or is exactly in the $\circ$-state. This can be done under the assumption that the input units are set to output either t^- or t^+. Under this assumption we find that there are only finitely many possible inputs for each hidden layer unit u, because there are only finitely many input nodes. Let $I_u := \{i_1, \ldots, i_n\}$ be the possible inputs for u. We split the set into $I_u^+ := \{i \mid i \in I_u \text{ and } i > i^\circ\}$ and $I_u^- := \{i \mid I \in i_u \text{ and } i < i^\circ\}$. Those inputs are considered active and inactive respectively, but are not necessarily within the corresponding activation ranges as introduced above. Now, we define the factor m_u as follows:

$$m_u^+ := \left|\frac{i^+}{\min(I_u^+) - i^\circ}\right| \qquad m_u^- := \left|\frac{i^-}{i^\circ - \max(I_u^-)}\right| \qquad m_u := \max\left(m_u^-, m_u^+\right). \tag{3.1}$$

If I_u^+ or I_u^- are empty, and hence $\min(I_u^+)$ or $\max(I_u^-)$ are undefined, we set m_u^+ or m_u^- to be 0. In the pathological case that both are empty, we find that the unit u is always in the $\circ$-state according to Definition 3.2.1. This implies that all incoming weights are 0 and the threshold is $-i^\circ$. For each non-pathological unit, we find after multiplying the weights and the threshold by m_u, that the unit is active (inactive) for all inputs in I_u^+ (I_u^-), or that it is in the $\circ$-state for the input i°.

This transformation can be applied to all direct successors of input units. It can also be applied recursively to every other unit in the network, because the network is finite and acyclic. After adapting all weights and thresholds of the network, we find that the network behaves exactly as the threshold networks constructed in the previous section. I.e., a hidden unit corresponding to some clause i is in the active state if and only if the body of that clause is true under the current interpretation. And furthermore, an output unit corresponding to some a is active if at least one hidden unit corresponding to some clause with head a is active.

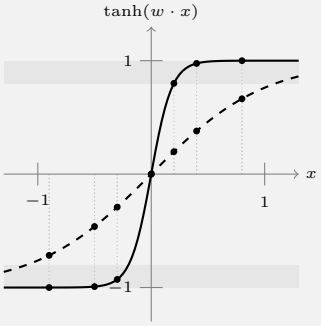
Unfortunately, the transformation described above is not feasible due to the exponential number of different input patterns that need to be generated to find $\min(I_u^+)$ and $\max(I_u^-)$. But it allows to presume that there is a network for any smooth activation function computing T_P-operators.

Fortunately, for some networks and some s^-, s° and s^+, we can compute $\min(I_h^+)$ and $\max(I_h^-)$ directly. This is possible, e.g., for the networks constructed in the previous section

Example 3.2.4 Let us consider a tanh-unit u with a threshold of 0 and a single incoming connection with weight 1 and let i^-, i° and i^+ be as in Example 3.2.1. Let the predecessor unit have the following possible outputs $\{-0.9, -0.5, -0.3, 0.0, 0.2, 0.4, 0.8\}$ (which coincide with the possible inputs for u as the weight is $w = 1$). Therefore, we find $I_u^- = \{-0.9, -0.5, -0.3\}$ and $I_u^+ = \{0.2, 0.4, 0.8\}$, with $\max(I_u^-) = -0.3$ and $\min(I_u^+) = 0.2$. This yields,

$$m_u^+ = \left|\frac{1.1}{0.2 - 0.0}\right| = 5.493 \qquad m_u^- = \left|\frac{1.1}{0.0 + 0.3}\right| = 3.662$$

and hence $m_u = 5.493$. The plot of the function, using dashed lines for $w = 1$ and solid lines for the function shape after multiplying the weight by m_u is show below. It also contains the input values together with the resulting activations:



$\mathbf{x}$	$\tanh(\mathbf{1} \cdot \mathbf{x})$	$\tanh(\mathbf{m_u} \cdot \mathbf{x})$
-0.9	-0.716	-0.999
-0.5	-0.462	-0.992
-0.3	-0.291	-0.929
0.0	0.000	0.000
0.2	0.197	0.800
0.4	0.380	0.976
0.8	0.664	0.999

Note that after multiplying the weight by $m_u = 5.493$, the function is scaled such that for all inputs $\neq i^\circ$, the activations are either above a^+ or below a^-.

and the threshold counterpart Θ_{-1}^{+1} of the tanh function. For $s^- = -1$, $s^\circ = 0$ and $s^+ = 1$, we find $\min(I_h^+) = s^+$ and $\max(I_h^-) = s^-$. This results from the fact, that all weights are actually integers and we use s^- and s^+ as input activations only. I.e., we can compute m_h for all hidden units by using $i^\circ = 0$, $\min(I_h^+) = s^+$ and $\max(I_h^-) = s^-$ in Equation (3.1):

$$m_h = \max\left(\left|\frac{i^+}{s^+ - i^\circ}\right|, \left|\frac{i^-}{i^\circ - s^-}\right|\right) = \max(|i^+|, |i^-|).$$

After multiplying all input to hidden layer weights with m_h, we find that their output is either above a^+, below a^-, or exactly a°. As the output layer units serve as disjunctions, we need to ensure that an output layer unit is active whenever at least one of its predecessor units with connection weight > 0 is active. Therefore, we compute m_o for all output units using $i^\circ = 0$, $\min(I_h^+) = a^+$ and $\max(I_h^-) = a^-$ as follows:

$$m_o = \max\left(\left|\frac{i^+}{a^+ - i^\circ}\right|, \left|\frac{i^-}{i^\circ - a^-}\right|\right) = \max\left(\left|\frac{i^+}{a^+}\right|, \left|\frac{i^-}{a^-}\right|\right).$$

And after multiplying all hidden to output layer weights with m_o, we find that their output is as well either above a^+, below a^-, or exactly a°. To avoid numerical problems in our implementations, we multiply m_h and m_o with some factor $m > 1$. After introducing Bipolar Sigmoidal networks formally, we show the equivalent behaviour with the bipolar threshold networks constructed in the previous section.

Remark 3.2.5 *Please note, that we use $a^- = -a^+$ in the following definition. This is not a real restriction, but simplifies the proofs a lot. All what follows should work with arbitrary values $a^- < a^+$ as well, but would be tedious to write down.*

Definition 3.2.6 (Bipolar Sigmoidal PropCore-Network) *Let P be a propositional logic program over $\mathcal{V}$, $0 < a < 1 \in \mathbb{R}$ and $1 < m \in \mathbb{R}$. We use $a^+ = a$, $a^- = -a$ as thresholds to define the (in-)activation ranges. Let $i^+ := \tanh^{-1}(a)$ and $i^- := \tanh^{-1}(-a)$ (i.e. $i^+ = -i^-$). Let the factors m_h and m_o be as follows:*

$$m_h = \max(|i^+|, |i^-|) = \tanh^{-1}(a) \qquad m_o = \max\left(\left|\frac{i^+}{a^+}\right|, \left|\frac{i^-}{a^-}\right|\right) = \frac{\tanh^{-1}(a)}{a}.$$

Let $\mathrm{tpcn}_{\mathcal{V}}^{-1,0,1}(P) = \left\langle \Theta_{t^-}^{t^+}, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}} \right\rangle$ be the threshold network constructed for P as introduced in Definition 3.1.3. The bipolar sigmoidal network $\mathrm{bspcn}_{\mathcal{V}}^{a}(P)$ is obtained from $\mathrm{tpcn}_{\mathcal{V}}^{-1,0,1}(P)$ as follows:

$$\begin{aligned}
\mathrm{bspcn}_{\mathcal{V}}^{a}(P) &:= \left\langle \tanh, \mathcal{U}_{\mathsf{inp}}, \omega'_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}'_{\mathsf{hid}}, \omega'_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}'_{\mathsf{out}} \right\rangle \text{ with} \\
\mathcal{U}'_{\mathsf{hid}} &:= \left\{ \mathsf{hid}_n(\theta') \mid \mathsf{hid}_n(\theta) \in \mathcal{U}_{\mathsf{hid}} \text{ and } \theta' = m \cdot m_h \cdot \theta \right\} \\
\mathcal{U}'_{\mathsf{out}} &:= \left\{ \mathsf{out}_n(\theta') \mid \mathsf{out}_n(\theta) \in \mathcal{U}_{\mathsf{out}} \text{ and } \theta' = m \cdot m_o \cdot \theta \right\} \\
\omega'_{\mathsf{inp}\blacktriangleright\mathsf{hid}} &: \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \to \mathbb{R} \\
&\quad (\mathsf{inp}_a, \mathsf{hid}_n) \mapsto m \cdot m_h \cdot \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}(\mathsf{inp}_a, \mathsf{hid}_n) \\
\omega'_{\mathsf{hid}\blacktriangleright\mathsf{out}} &: \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}} \to \mathbb{R} \\
&\quad (\mathsf{hid}_n, \mathsf{out}_a) \mapsto m \cdot m_o \cdot \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}(\mathsf{hid}_n, \mathsf{out}_a)
\end{aligned}$$

The following proposition shows the desired relation between the T_P-operator of a given program P and the function computed by the bipolar sigmoidal PropCore network constructed for P.

Proposition 3.2.7 *Let P be a logic program over $\mathcal{V}$ with $|\mathcal{V}| = m$, let $0 < a < 1 \in \mathbb{R}$ and $a^+ = a$, $a^- = -a$. Let $\mathrm{bspcn}_{\mathcal{V}}^{a}(P)$ be the corresponding* tanh*-PropCore-network and let $f_P : \mathbb{R}^m \to \mathbb{R}^m$ be the function computed by $\mathrm{bspcn}_{\mathcal{V}}^{a}(P)$. Then the following diagram commutes:*

$$\begin{array}{ccc}
I & \xrightarrow{T_P} & T_P(I) \\
{\scriptstyle \boldsymbol{\iota}_{\mathcal{V},|\cdot|}^{(-1,1)}}\downarrow & & \uparrow{\scriptstyle \kappa_{\mathcal{V},|\cdot|}^{(a^-,a^+)}} \\
x & \xrightarrow[f_P]{} & f_P(x)
\end{array}$$

I.e., we find that $T_P(I) = \kappa_{|\cdot|}^{(a^-,a^+)}\left(f_P\left(\boldsymbol{\iota}_{|\cdot|}^{(-1,1)}(I)\right)\right)$ holds for all $I \in \mathcal{I}_{\mathcal{V}}$.

> **Proof (sketch)** As in the proof of Proposition 3.1.4, this proof could be done in four steps. First (i), we need to show that a hidden unit hid_c, constructed for some clause $h \leftarrow$ Body, is active if and only if the input unit's activation pattern is a model for that body. Next (ii), we would show that an output unit out_a is active if and only if at least one hidden unit corresponding to a clause with head a was active at the previous time step. From (i) and (ii) it follows (iii) that after presenting some interpretation I to the network at time t, that the output unit's activation pattern at time $t+2$ corresponds to $T_P(I)$. Finally (iv), we conclude that $\boldsymbol{\iota}(T_P(I)) = f_P(\boldsymbol{\iota}(I))$ and $T_P(I) = \kappa_{|\cdot|}^{(a^-,a^+)}\left(f_P\left(\boldsymbol{\iota}_{|\cdot|}^{(-1,1)}(I)\right)\right)$. □

An algorithm to construct bipolar sigmoidal PropCore-networks follows directly from Definition 3.2.6, i.e. by using the algorithm from Figure 3.2 and changing the resulting network appropriately. It is given in Figure 3.3. As before, we can conclude that there is such a network for each logic program P, and that it can actually be constructed.

```
1  bspcn(A, M, V, Prog, Network) :-
2      tpcn(-1,0,+1, V, Prog, ffn(Ui, I2H, Uh, H2O, Uo, _)),
3
4      Mh := arctanh(A)*M,
5      Mo := Mh / A,
6
7      Uh2 := { unit(N, T2) | unit(N, T) in Uh, T2 := T*Mh },
8      Uo2 := { unit(N, T2) | unit(N, T) in Uo, T2 := T*Mo },
9
10     I2H2 := { S/T/Wm | S/T/W in I2H, Wm := W*Mh },
11     H2O2 := { S/T/Wm | S/T/W in H2O, Wm := W*Mo },
12
13     Network = ffn(tanh, Ui, I2H2, Uh2, H2O2, Uo2).
```

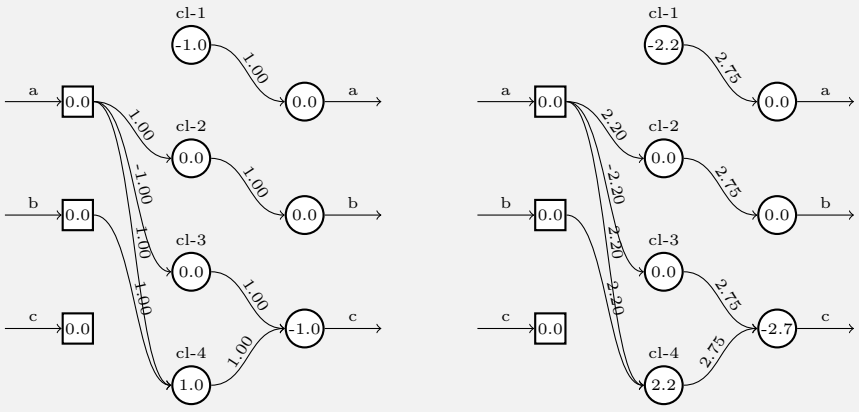
Figure 3.3: *Implementation for the construction of a behaviour equivalent network with units computing the hyperbolic tangent.*

Corollary 3.2.8 *For each propositional logic program P there exists a 3-layer feed-forward network with bipolar sigmodial units which computes T_P.*

Example 3.2.5 Using tanh as activation function and $a = 0.8$, we find:

$$m_h = \tanh^{-1}(0.8) \approx 1.1 \qquad m_o = \frac{\tanh^{-1}(0.8)}{0.8} \approx 1.375.$$

Those are multiplied by $m = 2$ and we obtain the following bipolar sigmodial PropCore-network:



The threshold-counterpart (see Example 3.1.2) is re-plotted on the left.

So far, we were concerned with the connectionist implementation of the T_P operator. In the previous section, we have shown that this operator can be implemented using threshold networks. And in this section, we extended those results to networks with smooth activation functions. But we did not yet discuss the iteration of the computation. I.e., the dynamics of the system if we feed back the results from the output layer back to the input layer. This is done in the following section.

3.3 Iterating the Computation of the Embedded Operators

To allow the iteration of the T_P-operator we connect the output units recurrently back to the input units. A recurrent connection is added such that every output unit out_a is connected to input unit inp_a with weight 1. Thus, the activations of the output layer are propagated back to the input layer. As above, we first discuss the case of threshold networks, and then their smooth counterparts.

Definition 3.3.1 (Recurrent Threshold PropCore-Network) *Let P be a logic program and let $\mathrm{tpcn}_{\mathcal{V}}^{t^-,t^\circ,t^+}(P) = \left\langle \Theta_{t^-}^{t^+}, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}} \right\rangle$ be the corresponding threshold network. The* recurrent PropCore-network *is defined as follows:*

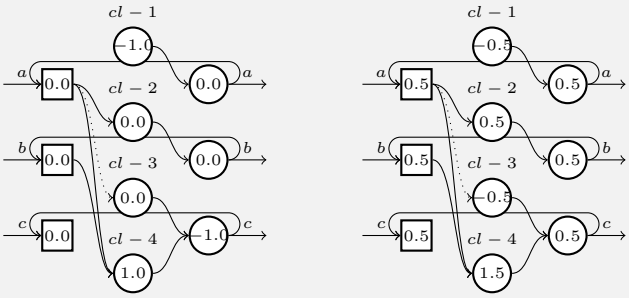
$$\mathrm{rtpcn}_{\mathcal{V}}^{t^-,t^\circ,t^+}(P) = \left\langle \Theta_{t^-}^{t^+}, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}}, \omega_{\mathsf{out}\blacktriangleright\mathsf{inp}} \right\rangle \quad \textit{with}$$

$$\omega_{\mathsf{out}\blacktriangleright\mathsf{inp}} : \mathcal{U}_{\mathsf{out}} \times \mathcal{U}_{\mathsf{inp}} \to \mathbb{R}$$

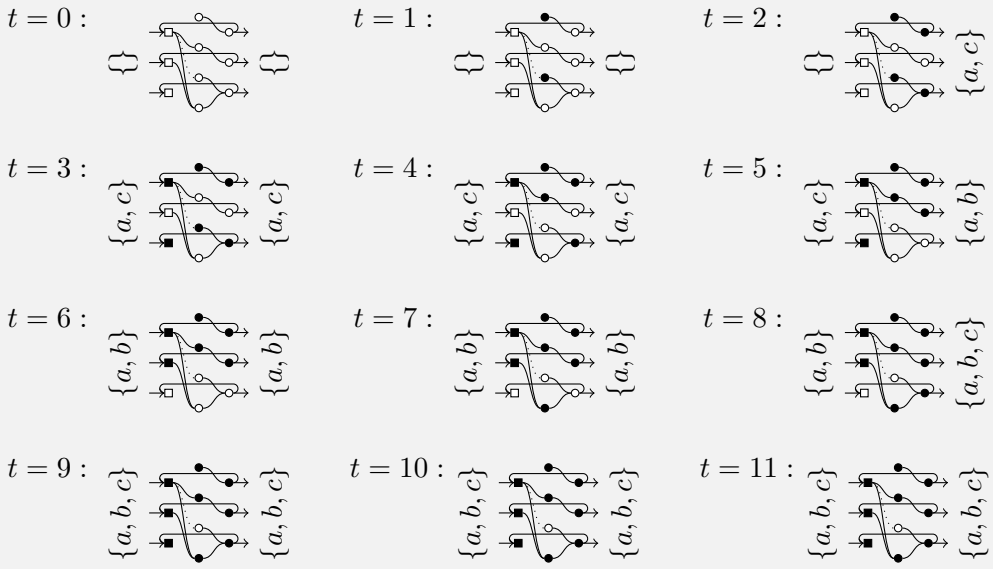
$$(\mathsf{out}_a, \mathsf{inp}_b) \mapsto \begin{cases} 1 & \textit{if } a = b \\ 0 & \textit{otherwise} \end{cases}$$

Example 3.3.1 The recurrent threshold PropCore-networks for our running example (on the left) for Θ_{-1}^{+1} in the middle and for Θ_0^1 on the right :

$(cl{-}1) \quad a.$
$(cl{-}2) \quad b \leftarrow a.$
$(cl{-}3) \quad c \leftarrow \neg a.$
$(cl{-}4) \quad c \leftarrow a \wedge b.$



The states of the networks for the first twelve timesteps are shown below. Please note that this network reaches a stable state already at $t = 9$.



It remains to be shown that the iteration of f_P and the iteration of T_P lead to the same result. For the threshold case, this means $\boldsymbol{\iota}^{-1}(f_P^n(\boldsymbol{\iota}(I))) = T_P^n(I)$ holds for all $n \geq 0$ and all $I \in \mathcal{I}_\mathcal{L}$.

Proposition 3.3.2 *Let* $\mathrm{rtpcn}_\mathcal{V}^{t^-,t^\circ,t^+}(P)$ *be defined as above. Then*

$$\boldsymbol{\iota}^{-1}(f_P^n(\boldsymbol{\iota}(I))) = T_P^n(I)$$

holds for all $n \geq 0$ *and all* $I \in \mathcal{I}_\mathcal{L}$.

Proof This equality follows immediately from Proposition 3.1.4, the bijectivity of $\boldsymbol{\iota}$ and the fact that f_P is a function to $\{t^-, t^+\}^{|\mathcal{V}|}$. □

We can define a recurrent network for a given bipolar sigmoidal network analogously to Definition 3.3.1. But unfortunately, we can not prove the correctness anymore, because Definition 3.2.6 relies on the fact that the input units activations are integers. This can not be guaranteed while iterating the computation, because f_P is a function to $\mathbb{R}^{|\mathcal{V}|}$. But we can enforce integer activation in the input layer by using threshold units there.

Definition 3.3.3 *Let* $\mathrm{bspcn}_\mathcal{V}^a(P) = \langle \tanh, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}} \rangle$ *be a bipolar sigmoidal network according to Definition 3.2.6. Then we define the* recurrent PropCore-network *as follows:*

$$\mathrm{rbspcn}_\mathcal{V}^a(P) = \left\langle \mathcal{U}_{\mathsf{inp}}, \Theta_{-1}^{+1}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \tanh, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}}, \tanh, \omega_{\mathsf{out}\blacktriangleright\mathsf{inp}} \right\rangle \quad \textit{with}$$

$$\omega_{\mathsf{out}\blacktriangleright\mathsf{inp}} : \mathcal{U}_{\mathsf{out}} \times \mathcal{U}_{\mathsf{inp}} \to \mathbb{R}$$

$$(\mathsf{out}_u, \mathsf{inp}_v) \mapsto \begin{cases} 1 & \textit{if } u = v \\ 0 & \textit{otherwise} \end{cases}$$

The following theorem states the correspondence between the iteration of the T_P-operator and the iteration of the network function. As required, both agree in the sense that for all $I \in \mathcal{I}_\mathcal{L}$ and all $n \geq 0$ we find $\kappa(f_P^n(\boldsymbol{\iota}(I))) = T_P^n(I)$.

Theorem 3.3.4 *Let* $\boldsymbol{\iota}$, κ *and* $\mathrm{rbspcn}_\mathcal{V}^{t^-,t^\circ,t^+}(P)$ *be defined as above. Then*

$$\kappa(f_P^n(\boldsymbol{\iota}(I))) = T_P^n(I)$$

holds for all $n \geq 0$ *and all* $I \in \mathcal{I}_\mathcal{L}$.

Proof This equality follows from Proposition 3.2.7, Lemma 3.2.4 and the fact that the threshold units in the input layer enforce the units to output either t^- or t^+. □

For completeness reasons, we restate the following result (compare Corollary 3.3 in [HHS04]). As shown in Corollary 2.3.23, acyclic programs have a unique supported model, which can be computed by iteration of the associated T_P-operator starting with an arbitrary interpretation. Furthermore, this iteration yields the unique fixed point after at most $n \times m$ applications of the T_P-operator, where n and m are the number of clauses and propositional variables occurring in the program, respectively. This allows to conclude that recurrent threshold networks constructed for such programs settle down to a stable state corresponding to the supported model.

Corollary 3.3.5 *Let P be an acyclic propositional logic program as introduced in Definition 2.3.20 and let N be a corresponding PropCore-network. Starting from an arbitrary initial state, N will always settle down to a unique stable state after at most $n \times m$ time-steps, with n being the number of clauses and m being the number of propositional variables occurring in P. The interpretation corresponding to the activation pattern of the output layer at $t = n \times m$, coincides with the unique supported model of P.*

3.4 Summary

In this chapter, we have shown how to embed propositional interpretations into vectors of real numbers, and how to construct a connectionist system for a given semantic operator, such that the corresponding network function and the embedded operator coincide. This has first been done for networks that apply the threshold function and then the constructions have been extended to sigmoidal activation functions. In the case of threshold networks, we followed in principle the ideas described by Steffen Hölldobler and Yvonne Kalinke in [HK94], but the constructions presented here allow for more flexibility. While extending the results to smooth activation functions, we have computed a factor for the weights to ensure the correctness. This was done similarly to [dGZ99], but again allows for more flexibility wrt. the activation function. Furthermore, the boundaries in the approach presented here can be set by the user, which allows a more fine grained control of the dynamics of the resulting network. Finally, we have discussed how to change the networks such that they compute not only a single application of the T_P-operator, but also its least model. This has been done by recurrently connecting the output layer back to the input layer.

As mentioned above, the approach discussed here generalised ideas presented in [HK94], [TS94] and [dGZ99]. For example, the networks constructed in [HK94] are Threshold PropCore-Network as introduced in Definition 3.1.3 using $t^- = 0.0$, $t^\circ = 0.5$ and $t^+ = 1.0$. Therefore, our constructions serve as a unification of all those approaches in the sense that those are instances of the method presented here.

The presented results and algorithms constitute the first part of the neural-symbolic cycle presented in Figure 1.1, i.e., the embedding of symbolic knowledge into a connectionist system. Concerning the classification scheme presented in Section 1.3, the results presented above, can be classified as follows:

Interrelation: The approach is an *integrated* one, i.e., the result is a connectionist system processing symbolic knowledge. The result is a *connectionist* system and we are not concerned with its biological plausibility. The symbolic knowledge is presented *locally* because a single input or output unit corresponds to a single proposition, and every hidden unit corresponds to one particular rule embedded into the network. The resulting architecture can be considered as *standard* in the area of artificial neural networks.

Language: On the symbolic side, we have been concerned with *propositional logic formulae*.

Usage: So far we have been concerned with the *representation* of symbolic knowledge with a connectionist system. And once this knowledge is embedded, we showed that *reasoning* is possible.

The following chapters are concerned with extensions towards *learning* and *extraction* capabilities. I.e., the final system covers the whole *usage* dimension.

The representation of proposition has been done by an identification with input and output units and by fixing activity ranges corresponding to true and false. Rules have been embedded into the network by adding a hidden unit and setting its threshold and weights such that it

becomes active if and only if all precondition of the rule are satisfied. This has been done for threshold units first and afterwards for units with smooth activation functions. The latter allow the application of standard learning algorithms like backpropagation. How this can be done in detail is discussed in the following chapter. Chapter 6 is concerned with the extraction of propositional rules from trained networks, and thus with closing the neural-symbolic cycle for propositional logic we started here.

4 An Application to Part of Speech Tagging

... where we discuss how to initialise an artificial neural network with grammatical background knowledge and apply it to the problem of part of speech tagging. After describing the application domain in Section 4.1, we discuss the neural-symbolic tagger in detail. In Section 4.3 some experimental results are discussed. We conclude this chapter with a summary in Section 4.4.

This chapter is partly based on [MBRH07] and on experiments performed together with Nuno Miguel Cavalheiro Marques.

Part of speech tagging (POS), also called syntactic word-class tagging, assigns grammatical tags (like noun, verb, etc.) to a word depending on its definition and its context [vH99]. POS tagging plays an important role in the area of language processing, because it is the first step before analysing the semantics of a given sentence.

Here, we show how to use an artificial neural network to learn the tagging function [ML01, Sch94], but instead of randomising the weights, we initialised the network using some grammatical background knowledge. This knowledge, after representing it as propositional rules, is embedded into the network, following Chapter 3. The embedding of background knowledge leads to better results, with respect to the generalisation ability and the speed of training.

The system presented below is not yet a full tagger, because much remains to be done to make it really comparable to other systems. But we show, that the learning of the underlying network can be improved by embedding some background knowledge. Therefore, this chapter should be understood as a proof of concept, that neural-symbolic systems can be used in the area of natural language processing. And to the best of our knowledge, this is the biggest problem ever tackled using a neural-symbolic approach.

4.1 Part of Speech Tagging

Many words are grammatically ambiguous, e.g., the same word can be used as noun or adjective (e.g., "past" can be an adjective, noun or adverb). Part of speech tagging, also called syntactic word-class tagging, is the process that assigns grammatical tags (like noun, verb, etc.) to a word depending on its context [vH99]. First POS systems were rule-based. But rule-based systems for tagging are hard to build and maintain by hand. Also most of the rules, instead of being associated with language syntax, are purely Heuristic [vH99].

Statistical and machine learning approaches use supervised classification, usually based on annotated text corpora, to solve this problem. An annotated text corpus, consists of large texts whose words are tagged by hand. For the experiments discussed in Section 4.3, we used the *Susanne* corpus with a re-encoded tagset [Sam95]. Example 4.1.1 shows one sentence of that corpus, together with the correct tags for each word.

Example 4.1.1 Below you find an annotated sentence from the Susanne corpus. The tags are shown as subscripts attached to every word and punctuation symbol.

> "$_{\mathsf{pto}}$ I$_{\mathsf{prp}}$ am$_{\mathsf{be}}$ not$_{\mathsf{adv}}$ prepared$_{\mathsf{v}}$ to$_{\mathsf{prep}}$ grant$_{\mathsf{v}}$ bail$_{\mathsf{n}}$ to$_{\mathsf{prep}}$ any$_{\mathsf{pri}}$ of$_{\mathsf{prep}}$ them$_{\mathsf{prp}}$ "$_{\mathsf{pto}}$,$_{\mathsf{pto}}$ said$_{\mathsf{v}}$ the$_{\mathsf{det}}$ magistrate$_{\mathsf{n}}$.

4.2 System Architecture

Following [Sch94], our system works as follows: The a-priory tag-probabilities of the word under consideration together with its p predecessors and s successors are fed into an appropriately set up artificial neural network. The output layer contains t units, each representing one possible tag. During the training, the network learns to output a value close to 1 for the correct tag and close to 0 for all other tags. To tag a word w we consider its context and based on this context we compute the most likely tag. The function from a word's context to its most likely tag is computed by a neural network as described below.

The general architecture of our system is shown in Figure 4.1. A dictionary is extracted from an annotated corpus as described in Section 4.2.1. An artificial neural network is constructed

from an initial grammar by first transforming the grammar into if-then-rules as shown in Section 4.2.2. Those rules are embedded into a suitable network (described in Section 4.2.3). The embedding is detailed in Section 4.2.4. Using the corpus and the dictionary we generate training-data suitable for our network as shown in Section 4.2.5. Finally, the improved network is used for tagging.

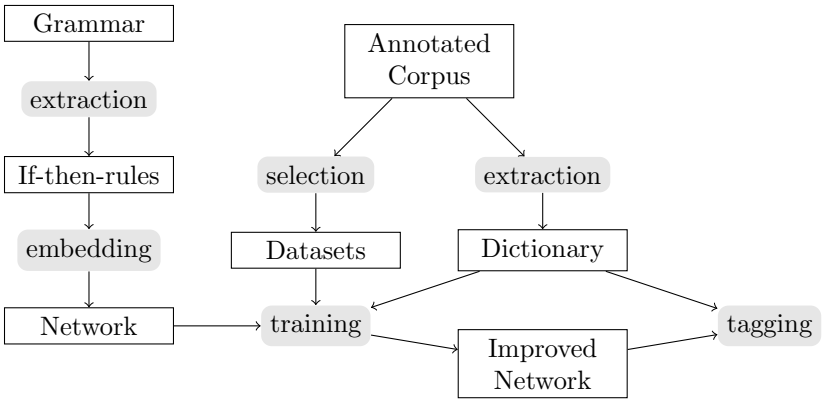


Figure 4.1: *General architecture of a neural-symbolic part of speech tagger.*

4.2.1 Extracting a Dictionary from a Given Text Corpus

Based on a given corpus, we can generate a dictionary. A dictionary contains every word of the corpus, together with its probability distribution over the tagset and is usually obtained by simply counting the occurrences. A small part of the dictionary for the Susanne corpus is shown in Example 4.2.1.

Example 4.2.1 The table below shows a part of the dictionary extracted from the Susanne corpus. As mentioned above, the dictionary is a probability distribution for known words over tags.

Word	**Probability distribution**		
right	P_{adj}=42%	P_{adv} =25%	P_{n} =33%
following	P_{adj}=40%	P_{prep}=25%	P_{v} =35%
further	P_{adj}=50%	P_{adv} =42%	P_{v} = 8%
beat	P_{adj}=50%	P_{n} = 8%	P_{v} =42%
opening	P_{adj}=50%	P_{n} =20%	P_{v} =30%
round	P_{adj}=17%	P_{adv} =50%	P_{v} =33%
outside	P_{adj}=17%	P_{adv} =33%	P_{prep}=50%
grant	P_{n} =75%	P_{v} =25%	

In the sentence shown in Example 4.1.1, *grant* is correctly tagged as a verb, despite the dictionary saying that it is probably a noun (75% versus 25% for verb). This illustrates, that context information is indeed necessary.

4.2.2 Extracting Rules from a Grammar

In the approach presented here, we did not embed the grammar itself, but rather the resulting context rules. Considering the grammar shown in Example 4.2.2, we find a proper noun (pn) whenever there is a punctuation mark (pto) to the left and a verb (v) to the right. This is due to the fact, that one possibility of expanding a sentence S yields the sequence [pto, pn, v, pto].

To generate all context information implicit in the grammar, we generated possible sentences by unfolding the grammar to its terminal symbols. Unfolding the grammar from Example 4.2.2 five times results in 88 different sentences as presented in Example 4.2.3. From those sentences we generated 35 different 3-tuples, which can be transformed into propositional rules as shown in Example 4.2.4. The tuples are generated by simply taking three subsequent terminal symbols from the constructed sentences. From all those triples, we can select the most frequent ones, because they have very likely the biggest impact. A triple $[l, c, r]$ is converted into the corresponding context rule $c \leftarrow l_{-1} \wedge r_{+1}$, stating that the symbol left to c (index = -1) must be an l and the symbol to the right (index=+1) must be r.

Example 4.2.2 A (very) simple grammar for English. Non-terminal symbols are shown in capital letters, while terminal symbols are shown as terminal.

$$
\begin{aligned}
S &\rightarrow \mathsf{pto}, NP, VP, \mathsf{pto} & VP &\rightarrow \mathsf{v} \\
NP &\rightarrow \mathsf{pn} & VP &\rightarrow \mathsf{v}, NP \\
NP &\rightarrow \mathsf{det}, \mathsf{n}, OPTREL & OPTREL &\rightarrow \mathsf{that}, VP \\
NP &\rightarrow \mathsf{det}, \mathsf{adj}, \mathsf{n}, OPTREL & OPTREL &\rightarrow \varepsilon
\end{aligned}
$$

Example 4.2.3 Some syntactically correct "skeletons" of English sentences according to the grammar from Example 4.2.2. They are obtained by unfolding the grammar five times to its terminal symbols.

$$
\begin{aligned}
&[\mathsf{pto}, \mathsf{det}, \mathsf{adj}, \mathsf{n}, \mathsf{v}, \mathsf{det}, \mathsf{adj}, \mathsf{n}, \mathsf{pto}], \\
&[\mathsf{pto}, \mathsf{det}, \mathsf{adj}, \mathsf{n}, \mathsf{v}, \mathsf{pto}], \\
&[\mathsf{pto}, \mathsf{det}, \mathsf{n}, \mathsf{v}, \mathsf{det}, \mathsf{n}, \mathsf{pto}], \\
&[\mathsf{pto}, \mathsf{det}, \mathsf{n}, \mathsf{v}, \mathsf{pto}], \\
&[\mathsf{pto}, \mathsf{pn}, \mathsf{v}, \mathsf{pto}], \\
&\ldots
\end{aligned}
$$

Example 4.2.4 Some 3-tuples obtained from the sequences shown in Example 4.2.3 together with their corresponding context-rules.

$$
\begin{aligned}
[\mathsf{pto}, \mathsf{det}, \mathsf{n}] &\rightsquigarrow \mathsf{det} \leftarrow \mathsf{pto}_{-1} \wedge \mathsf{n}_{+1} \\
[\mathsf{det}, \mathsf{n}, \mathsf{that}] &\rightsquigarrow \mathsf{n} \leftarrow \mathsf{det}_{-1} \wedge \mathsf{that}_{+1} \\
[\mathsf{n}, \mathsf{that}, \mathsf{v}] &\rightsquigarrow \mathsf{n} \leftarrow \mathsf{n}_{-1} \wedge \mathsf{v}_{+1}
\end{aligned}
$$

4.2.3 An Artificial Neural Network for Part-of-Speech Tagging

As mentioned above, we consider context vectors to tag a given word: the tag-probabilities of the word and its p predecessors and s successors are fed into the network. I.e., the input layer contains $t \cdot (p+1+s)$ units, with t being the number of different tags. For example the input unit "adj_{-1}" is activated with the probability of the word on position -1 to be an adjective (adj). In the output layer, we used t units, each representing one possible tag.

To tag a word w, we consider its context, represented as a probability distribution. Depending on those probabilities, we compute a likelihood-vector over tags for w and pick the maximal element as result. The function from input-probabilities to output-likelihoods ($f : \mathbb{R}^{(p+1+s)\cdot t} \to \mathbb{R}^t$), is learned and computed by a neural network as depicted in Figure 4.2.

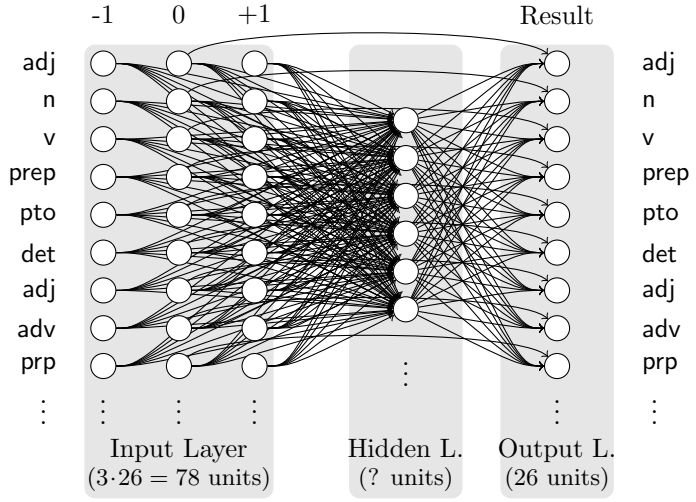


Figure 4.2: *Network architecture for part-of-speech tagging.*

4.2.4 Embedding Rules into a Network

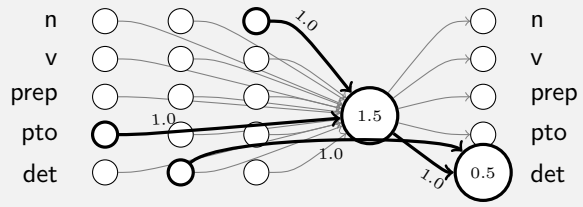
To embed a given rule into the network, we follow Chapter 3. I.e., we generate a hidden layer unit for each rule such that this unit will be active if and only if the preconditions of the rule are met. For example, to embed the rule $\mathsf{det} \leftarrow \mathsf{pto}_{-1} \wedge \mathsf{n}_{+1}$, we generate a hidden layer unit u with threshold 1.5 and add connections from pto_{-1} and n_{+1}, both weighted 1. This ensures that u is active if and only if pto_{-1} and n_{+1} are both active. Finally, u is connected to the unit det in the output layer with weight 1. The thresholds of all output units are set to 0.5.

Additionally, we added short-cut connections from each offset-0-unit in the input layer with the corresponding unit in the output layer. For example, the input unit det_0 is connected with the output unit det. These short-cuts were added, because in most cases the vote of dictionary and the correct result agree. Therefore, the network has to learn the exceptions only. Example 4.2.5 shows the result of embedding the rule $\mathsf{det} \leftarrow \mathsf{pto}_{-1} \wedge \mathsf{n}_{+1}$.

4.2.5 Generation of Training Data and Training the Network

Using the corpus and the extracted dictionary, we can generate training samples to train the resulting network as follows. Using a sliding window of the context size, we select a portion of the corpus. For all words of the window we take the probability distribution over tags as contained in the dictionary. This probability "matrix" forms the input. We construct an

Example 4.2.5 The result after embedding the rule $\mathsf{det} \leftarrow \mathsf{pto}_{-1} \wedge \mathsf{n}_{+1}$ together with the default-rule $\mathsf{det} \leftarrow \mathsf{det}_0$. The numbers denote weights and biases, respectively. All other weights are set to 0.



output vector containing 0 everywhere but 1 at the position of the correct tag. Part of the training data is shown in Example 4.2.6. Not the input and output vectors themselves are shown, but the more readable probability distributions.

Example 4.2.6 The table below contains some training samples for a window width of three. I.e., for the words to the left and to the right with respect to the word under consideration we take the probabilities from the dictionary. Those probabilities are then embedded into a vector of length $26 \cdot 3 = 78$ by setting the entries of this vector to the corresponding probabilities.

Word	$\mathbf{P_{-1}}$	$\mathbf{P_0}$	$\mathbf{P_{+1}}$	$\mathbf{P_{out}}$
to	$P_{\mathsf{pct}} = 0.99$	$P_{\mathsf{prep}} = 0.99$	$P_{\mathsf{n}} = 0.39 P_{\mathsf{v}} = 0.59$	$P_{\mathsf{prep}} = 0.99$
grant	$P_{\mathsf{prep}} = 0.99$	$P_{\mathsf{n}} = 0.39 P_{\mathsf{v}} = 0.59$	$P_{\mathsf{n}} = 0.97$	$P_{\mathsf{v}} = 0.99$
bail	$P_{\mathsf{n}} = 0.39, P_{\mathsf{v}} = 0.59$	$P_{\mathsf{n}} = 0.97$	$P_{\mathsf{prep}} = 0.99$	$P_{\mathsf{n}} = 0.99$
to	$P_{\mathsf{n}} = 0.97$	$P_{\mathsf{prep}} = 0.99$	$P_{\mathsf{wh}} = 0.99$	$P_{\mathsf{prep}} = 0.99$

After constructing a network by embedding grammatical background knowledge, we trained it using standard backpropagation in the *Stuttgart Neural Network Simulator (SNNS)* [Zel92].

4.3 Experimental Evaluation

The Susanne corpus contains 156.610 tagged words. As shown in Table 4.1, we split this set into different sub-corpora. $\approx 1/5$ is used as test set to compare the different tagging performances, $\approx 1/5$ is used for validation during the training to avoid overfitting phenomena and the remaining samples were used for training and dictionary building. We used three different subsets of the training set in our experiments, which are called *full*, *small* and *tiny* training set. Neither the test set nor the validation set have been used while computing the dictionary and during the training.

As a first experiment, we computed the mean square error (MSE) of different networks trained for the three training sets described above. The MSE is computed for the training and validation sets. As soon as the error on the validation set increases, we stop the training. Afterwards, the precision over the test is computed and used to actually evaluate the network.

A "NeSy"-network has been initialised using the 11 most frequent rules. This network is compared to a randomised copy. The results are shown in Figure 4.3.

The networks were trained on the full, small and tiny data-set and the error on the validation set was computed. On the full data set, the randomised network was the best, but on the small

Size	Usage
31.282	Testset (to compare the tagging performance)
31.305	Validation (used to avoid overfitting during training)
94.023	Training set *full*
9.424	Training set *small*
2.817	Training set *tiny*

Table 4.1: *Sub-corpora of the Susanne corpus used in our experiments.*

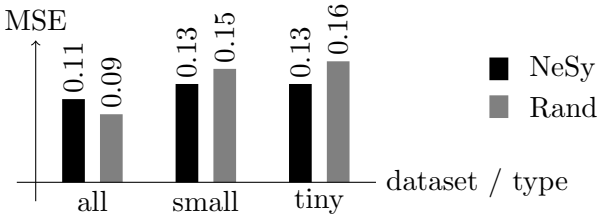


Figure 4.3: *Comparison of MSE after training an initialised and a purely randomised network.*

and tiny sets, the "NeSy"-network outperforms the randomised one. This is probably due to the fact that the rules were retained and are actually needed to achieve good results on the validation set, but could not be learned from the examples provided in the smaller data-sets.

Figure 4.4 shows a typical run for the small and tiny training corpus. The initialised network achieves very good results on the training corpus from the very beginning. But the randomised network gets better after only a few iterations. Nevertheless, the "NeSy"-network remains better on the validation corpus (which is actually the important one). This means that the initialised network generalises better to unseen samples.

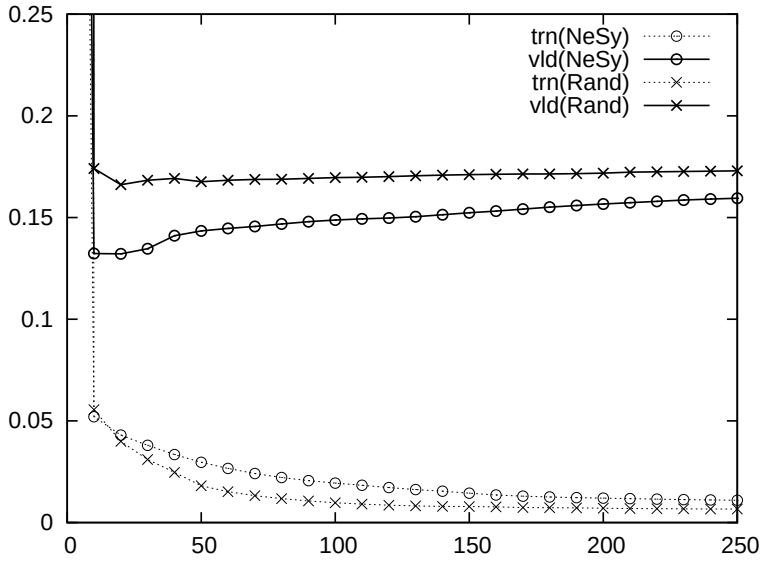


Figure 4.4: *The evolution of the error on training and validation set: the plot contains the first* 250 *iterations of training and shows the values for the "NeSy"- and the randomised network.*

4.4 Summary

In this chapter we discussed an application of a neural-symbolic system to part of speech tagging. The task is to assign grammatical tags to a word, depending on its definition and its context. Until now, two alternative approaches have been used for POS-tagging: On the one hand, there are rule based systems, which require a lot of work in system development and adaptation. On the other hand, there are supervised machine learning approaches, that require a lot of learning samples.

The proposed new approach conjoins both ideas. It is based on a neural network learning the tagging function. But instead of randomising the weights, we initialised the network using rules that express grammatical background knowledge.

We thus showed that neural-symbolic integration can be successfully applied in the context of POS tagging. Furthermore, we showed that the embedding of generic background knowledge is particularly helpful if only limited training data is available. It suffices to embed generic rules into a network which is then trained using only few examples to achieve good results. To the best of our knowledge, this is the biggest problem ever approached with neural-symbolic methods and the first application to natural language processing and text mining.

We believe that the area of natural language processing is particularly interesting for the field of neural-symbolic integration, because for most languages a considerable amount of grammatical rules and of annotated text corpora exists. Furthermore, it seems that the tagging problem itself is a problem that can be solved using neural networks. Most current systems are either based on rules or on neural networks, and both approaches are comparable with respect to their performance. We hope that taking the best of both worlds may lead to a better system.

However, much remains to be done. As mentioned above, the system is not yet a full tagger. Therefore, it is hard to really judge its performance. We only showed that the learning of the network can be improved. But the effect on the overall tagging performance needs to be evaluated.

5 Connectionist Learning and Propositional Background Knowledge

... where we show how to incorporate propositional knowledge into the training process. The approach presented here is based on the repeated insertion of error-correcting rules during the training. After showing how to embed propositional rules into connectionist systems, we focus now on the training of those systems with respect to propositional background knowledge.
In this section, we show how rules can be embedded repeatedly during training. We discuss the general methodology and a small experiment to verify the claims. After presenting the general idea, we discuss a simple application domain and the implementation of the ideas therein. Finally we discuss some experimental results.

This chapter is partly based on [BHM08] where we showed how to guide back-propagation by inserting error correcting rules.

Artificial neural networks are a very powerful tool to learn from examples, in particular for high dimensional and noisy data. Standard feed-forward networks together with backpropagation have been successfully applied in many domains. However, in most domains, at least some background knowledge in form of symbolic rules is available. And this knowledge can not be used easily.

Here, we discuss how rules can be embedded repeatedly during training. This helps to escape local minima which are the main problem while using standard learning schemes. The following observations have triggered this research:

- Very general rules, i.e., those that cover many training samples, are quickly acquired by backpropagation. Therefore, the embedding of those rules does not lead to a big improvement.
- Very specific rules that are embedded prior to the training process are very likely be overwritten by newly learned rules.
- We can analyse the errors made by the network during and after the training to obtain correcting rules.

5.1 Integrating Background Knowledge into the Training Process

The approach presented here is based on the repeated modification of the network during the training. After a number of training cycles, the errors made by the network are analysed. This analysis yields a rule that can be used to correct some of the errors. We then embed the rule into the network and continue with the next epoch as depicted in Figure 5.1. This helps the network to escape local minima during the training. I.e., the approach works as follows:

1. Initialise the network.
2. Repeat until some stopping condition is satisfied:
   a) Train the network for a given number of cycles.
   b) Analyse the errors of the network to obtain correcting rules.
   c) Embed the rule(s) into the network.

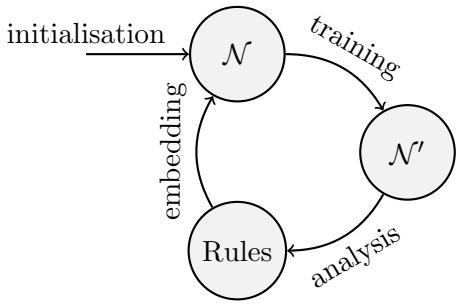


Figure 5.1: *The* rule-insertion cycle *for the integration of symbolic knowledge into the connectionist training process.*

This approach yields improved results on a simple classification task as shown in the following section.

5.2 A Simple Classification Task as Case Study

The "Tic-Tac-Toe Endgame Data Set" of the UCI Machine Learning Repository [AN07] is used for our experiment. The data set contains all 958 possible board configurations that can be reached while playing Tic-Tac-Toe starting with player X. Each configuration contains a description of the board and is classified as win or no-win for player X. A player wins the game, if he manages to place 3 of his pieces in a line. The board contains 3×3 cells and each cell can be marked by either x or o, or can be blank (b). Three possible boards are shown in Example 5.2.1. The goal of the Tic-Tac-Toe classification task is to decide whether a given board is a win-situation for player X or not.

Example 5.2.1 Three board configurations:

	Win for player X	**Draw**	**Win for player O**
Sample	$[x, b, o, o, x, o, x, b, x]^+$	$[x, x, o, o, o, x, x, o, x]^-$	$[x, o, x, o, o, b, x, o, x]^-$
Board	x ‧ o o x o x ‧ x	x x o o o x x o x	x o x o o ‧ x o x

5.2.1 The Connectionist Classifier

A simple 3-layer feed-forward network is used with tanh as activation function in all units. It contains $9 \times 3 = 27$ units in the input layer, that is three for each of the nine position on the board, corresponding to the three possible states. For each of this triples we use an 1-out-of-3 activation scheme. If a cell contains an "x" the x-unit for this cell is activated by setting the output to 1, while the o and b-units are set to -1. Initially, the network contains one hidden and one output unit. A board state is fed into the network by activating the appropriate input units. This activation pattern is propagated through the network and the networks decision can be read from the output unit. If its activation is greater than 0 the network evaluates the board as a win for player X, and as a no-win otherwise. Figure 5.2 shows the network used in our experiments.

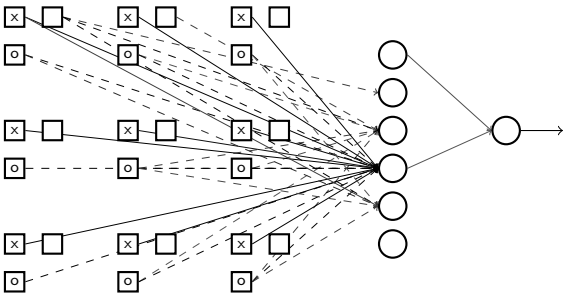


Figure 5.2: *A 3-layer fully connected feed-forward network with 27 input (rectangular), 6 hidden (cyclic) and a single output (cyclic) unit to classify Tic-Tac-Toe boards: The 27 input units are arranged like the board itself. Positive connections are shown as solid lines, while negative ones dashed. The width corresponds to the strength and small connections are omitted.*

The network is initialised with a single hidden unit and all connections as well as the bias of the units were randomised to values between -0.2 and 0.2. It is trained using standard

backpropagation in the *Stuttgart Neural Network Simulator (SNNS)* [Zel92]. The learning rate is set to $\eta = 0.1$. During each cycle, all samples are presented 100 times to the network.

5.2.2 Analysis of the Errors to Obtain Correcting Rules

After a full training cycle, all samples are presented again to the network and the network's output is compared to the desired values. All samples for which the difference between the computed and the desired output is greater than 0.5 are collected and grouped into positive and negative samples. We continue with the bigger set by constructing a template that is as accurate as possible and at the same time as general as possible. This is done by repeatedly selecting a cell and a value such that most erroneous samples agree on this. This is a variant of the "Learn one rule"-algorithm as used in sequential covering algorithms, but considers only one class [Mit97]. The following rule is constructed from 99 wrongly classified sample:

$$[\mathsf{b}_{13}, \mathsf{b}_3, \mathsf{o}_{23}, \mathsf{x}_{43}, \mathsf{x}_{34}, \mathsf{x}_{70}, \mathsf{o}_7, \mathsf{b}_1, \mathsf{b}_1] \mapsto +.$$

The subscripts show the percentage of the 99 samples which agree on the value. E.g., 13% of those samples have an empty upper left corner and 34% an x in the centre. We construct the rule template by ignoring all entries with a support $< 20\%$. This results in the following rule

$$[?, ?, \mathsf{o}, \mathsf{x}, \mathsf{x}, \mathsf{x}, ?, ?, ?] \mapsto +, \tag{5.1}$$

which actually covers 38 of the samples.

5.2.3 Embedding Rules into the Network

The rules constructed above, are embedded into the network following the ideas of the *Core Method* as discussed in Chapter 3. A new hidden unit is inserted into the network and the connections and the bias are set up such that this unit becomes active if and only if the input of the network coincides with the rule's precondition.

Let us use the rule $[?, ?, \mathsf{o}, \mathsf{x}, \mathsf{x}, \mathsf{x}, ?, ?, ?] \mapsto +$ to exemplify the idea. For all entries different from ?, a connection with weight ω from the input unit corresponding to the value and connections with weight $-\omega$ from the other two units are created. E.g., the x-unit for the central cell is connected to the new hidden unit with weight ω, while the corresponding o and b-units are connected with weight $-\omega$. All connections from input units for cells marked ? in the template are initialised to 0.0. The connection to the output unit is set to ω for positive rules and to $-\omega$ otherwise. The result is shown in Figure 5.3. Finally, all connections are slightly disturbed by adding some small random noise from $[-0.05, 0.05]$. The parameter ω is set to different values to study the impact of the embedded rules in the following section.

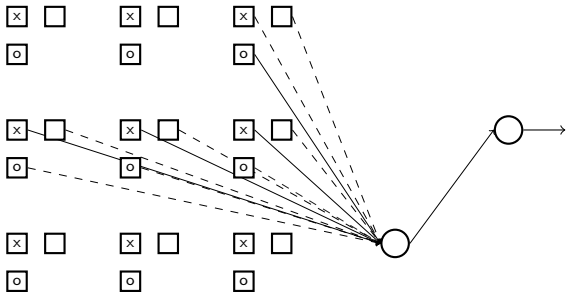


Figure 5.3: *The embedding of the rule* $[?, ?, \mathsf{o}, \mathsf{x}, \mathsf{x}, \mathsf{x}, ?, ?, ?] \mapsto +$*: Connections are depicted as described in Figure 5.2.*

5.3 Evaluation

To evaluate the method, we vary the value of $\omega \in \{0.0, 0.5, 1.0, 2.0, 5.0\}$. By setting it to 0.0, we can emulate the simple addition of a free hidden unit during the training, i.e., the new unit is initialised randomly. Therefore, this case should be the baseline for our experiments. Small values for ω would push the network only slightly into the direction of a rule, while a big value of 5.0 would strongly enforce the rule. In particular, if a rule was embedded with $\omega = 5.0$, then as soon as the corresponding hidden unit becomes active it's output will very likely dominate the sum of the outputs of the other hidden units. This is due to the fact that its activation of approximately 1.0 is propagated through a connection with a weight of 5.0 or -5.0 to the output unit. Therefore, the output unit simply follows the embedded rule. Consequently, as a rule of thumb, only rules where one is absolutely certain should be embedded with large ω.

Figure 5.4 shows the result of the experiment. It shows the mean squared error over time for the different settings of ω. For each value, we repeated the experiment 50 times and showed the averages. It also shows a zoom into the lower right corner, i.e., into the final cycles. Here, not only the averages, but also the standard deviation is shown. There are a few points to notice in those plots, which are discussed in more detail below:

- For $\omega = 0$ the network stops to improve at some point, even if units are added.
- For $\omega > 0$ the network outperforms the network where ω was set to 0 in the later cycles.
- For $\omega = 5$ the network did not learn very well.

Due to local minima in the error surface, backpropagation can get stuck in non-optimal solutions. Usually this is the case if there are errors on only few samples, because backpropagation follows in principle the majority vote. This probably also happens in our experiments. For $\omega = 0$, the network fails to improve further and remains at the 0.01-level. The method proposed here can help to solve this problem by inserting a unit which covers those few samples. Thus, we allow backpropagation to jump to better solutions.

The bad performance for $\omega = 5$, is due to the above mentioned fact about big values for ω and the fact that our rules are not necessarily correct. It happened several times in our experiments that the same (only partially correct) rule was embedded again and again. E.g., the rule $[?, ?, \mathsf{x}, ?, ?, ?, \mathsf{x}, ?, ?] \mapsto +$ was embedded, because most of the positive samples on which there were errors agreed on those values. But at the same time there are 42 negative samples with those values. Because the network blindly follows the rule, those negative samples are wrongly classified afterwards. During the training phase, the network basically unlearns the rule but could not improve on the original wrong positive samples. And thus, the same bad rule is embedded again and again. This problem could be solved using e.g., tabu-lists preventing the re-insertion. But we believe that better methods for the generation of rules should be used, which constructs only valid rules.

There is another finding in our experiments which is worth to be mentioned. E.g., the rule (5.1) from above covers only 38 samples, but the number of errors of the network on the training data went down from 99 to 26, i.e., inserting the rule apparently helps to better classify 73 samples. This is due to the better starting point provided by the insertion of the rule, that helps the network to correct errors more efficiently.

5.4 Summary

In this chapter, we have discussed a new scheme for the incorporation of background knowledge into the training process. It is based on an analysis of the errors made by the network after

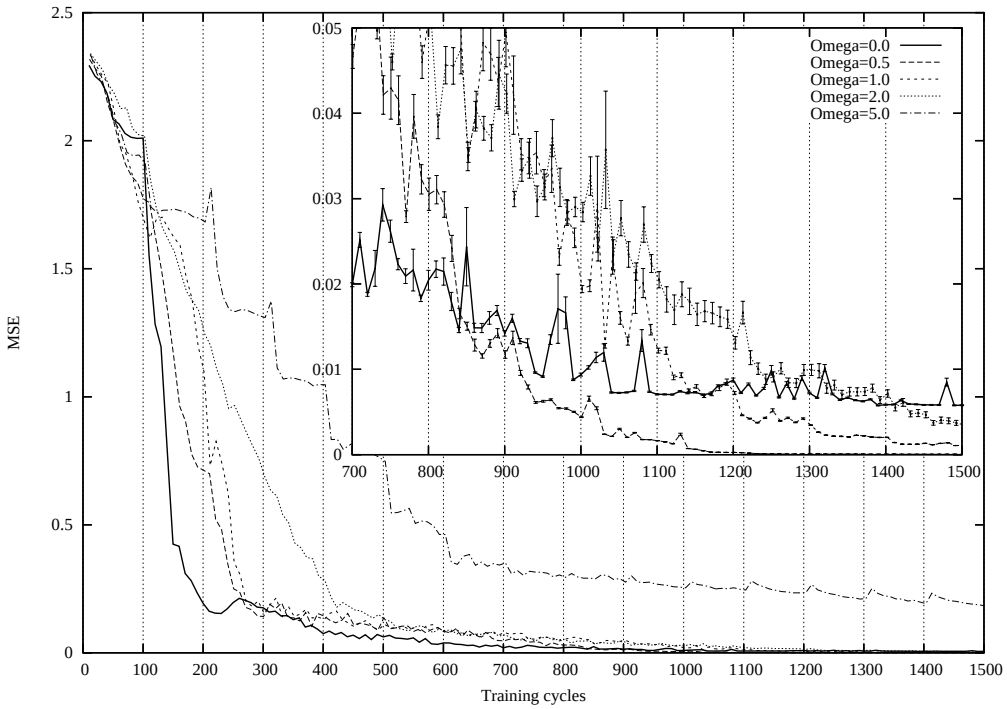


Figure 5.4: *The development of the mean squared error over time for different values of ω (averaged over 50 runs). The vertical lines show the time-points where rules have been inserted (every 100 cycles). The zoom in the upper right corner shows also the standard deviation over all 50 runs.*

training. Using the methods described in Chapter 3 it is possible to insert an error-correcting rule into the network.

We have also discussed a first experiment to support the claim that we can positively influence the training process of a feed-forward neural network by incorporating knowledge in the form of error-correcting rules into the training process. This is a first attempt to solve the challenge-problem number 4 presented in Section 1.4, i.e., the improvement of established learning algorithms using symbolic rules. Here, the rules were obtained from wrongly classified examples after an epoch of training the network using backpropagation. But other methods are possible as well.

The main problem remains to find good rules. Nuno Miguel Cavalheiro Marques and me have also applied ID3 for rule learning on all available data prior to the training and embedded the rules with big support. But we found that connectionist systems learnt such (very general) rules easily using backpropagation. The main problems are usually related with infrequent data patterns. Therefore, the analysis of the errors as proposed here is better than an a-priory generation of rules. But the problem of finding good rules remains. In this section, we have used a very naive approach to construct the correcting rules, but more powerful methods like ID3 or others could improve the performance even more. But the method of choice depends probably a lot on the application domain, and its choice is actually one of the points where background knowledge about the domain can be incorporated into the training process. Alternatively, error-analysis and rule creation could be done by a human expert. One should observe that this expert is only required once the network has learnt most of the rules. I.e., the human intervention is done only for the difficult cases and thus the expert does not need to explicitly state the simple rules.

Here, we have not used separate training and validation sets, because we wanted to study the improvement during learning. But as mentioned above, rule insertion can also lead to better generalisation. This is again very much dependant on the quality of the rules. If the rules generalise, their embedding also leads to a better generalisation of the neural network.

We have also found that rule insertion improves results beyond the knowledge directly expressed in rules. Indeed, similarly to work done when trying to improve neural networks by pruning irrelevant weights, rule insertion can also, indirectly help in the task of overriding irrelevant information. We believe this relation should be made clear in further work. Indeed the relevance of magnitude based pruning methods is well known in methods such as the ones related with optimal brain damage [LDS$^+$90]. More advanced training algorithms can also be used. E.g., R-Prop as reported in [JBN04] might be used to further increase speed and accuracy.

The results described here present only a starting point. But the methodology can be developed into a full fledged training paradigm that allows to incorporate domain-dependent background knowledge in a concise way.

6 Extracting Propositional Rules from Connectionist Systems

... where we extract rules from a feed-forward network. First, we define the rule extraction problem formally. In Section 6.2 to 6.5, we present the basic ideas of a new decompositional approach to it – the CoOp-approach. It is based on the internal representation using binary decision diagrams. Then, in Section 6.6, we discuss the incorporation of integrity constraints to refine the results and in Section 6.7 we show how to finally extract logic programs. We finish this chapter with an evaluation and a summary in Section 6.8 and 6.9, respectively.

This chapter is partly based on [BHME07], [Bad09] and on many fruitful discussions with Valentin Mayer-Eichberger.

During the training process, the networks acquire knowledge. Unfortunately, this learnt knowledge is hidden in the weights associated to the connections and humans have no direct access to it. One goal of rule extraction is the generation of a human-readable description of the output units behaviour with respect to the input units. Usually, the result is described in form of if-then rules, giving conditions that activate (or inactivate) a given output unit. Rule extraction from connectionist systems is still an open research problem, even though a number of algorithms exists. For an overview of different approaches we refer to [ADT95] and [Jac05]. In principle, extraction techniques can be divided into *pedagogical* and *decompositional* approaches. While the first conceives the network as a black box, the latter decomposes the network, constructs rules describing the behaviour of the simpler parts, and then re-composes those intermediate results.

6.1 The Rule Extraction Problem for Feed-Forward Networks

Intuitively, the rule extraction problem for feed-forward threshold networks is the search for a logic formula describing the behaviour of a given unit within the network. Because threshold units can be in two states only, we identify with every unit a propositional variable:

Notation 6.1.1 *Let A be a threshold unit. By an abuse of notation, we use A as a propositional variable and define it to be* true *iff the unit is active. We furthermore use $\bar{A}$ as the negation of A.*

Using this notation, we can represent the state of the input units $\mathcal{U}_{\mathsf{inp}}$ of a given threshold network $\langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ as interpretations over $\mathcal{U}_{\mathsf{inp}}$.

Definition 6.1.2 (Network Input) *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a given threshold feed-forward network with $|\mathcal{U}_{\mathsf{inp}}| = n$. Then we define the corresponding bijection $|\cdot|_{\mathcal{N}}$ between input nodes and $\{1, \ldots, n\}$ as follows:*

$$\begin{aligned} |\cdot|_{\mathcal{N}} : \mathcal{U}_{\mathsf{inp}} &\to \{1, \ldots, n\} \\ u &\mapsto i \text{ with } u \in_i \mathcal{U}_{\mathsf{inp}} \end{aligned}$$

Let $I \subseteq \mathcal{U}_{\mathsf{inp}}$ be an interpretation over $\mathcal{U}_{\mathsf{inp}}$. Using the bijection $|\cdot|_{\mathcal{N}}$ and the propositional embedding from Definition 3.1.1, we obtain the corresponding network input $\iota_{\mathcal{N}}(I)$ as follows:

$$\begin{aligned} \iota_{\mathcal{N}} : \mathcal{P}(\mathcal{U}_{\mathsf{inp}}) &\to \mathbb{R}^n \\ I &\mapsto \boldsymbol{\iota}_{|\cdot|_{\mathcal{N}}}^{(t^-, t^+)}(I) \end{aligned}$$

We can now define a propositional representation of some non-input unit A. Such a representation is a formula F over variables corresponding to input units such that networks input corresponding to models of the formula activate the unit A. I.e., after propagating $\iota_{\mathcal{N}}(I)$ through the network, we find A to be active. All interpretations which are no model for F should turn the unit inactive.

Definition 6.1.3 (Propositional Representation) *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a feed-forward network of threshold units, let f_o be as introduced in Definition 2.5.15 and let $A \in \mathcal{U}$ be some unit in $\mathcal{N}$. A propositional formula F over $\mathcal{U}_{\mathsf{inp}}$ is called a* propositional representation *of A iff for all interpretations I we find*

$$f_o(A, \iota_{\mathcal{N}}(I)) = \begin{cases} t^+ & \text{if } I \models F \\ t^- & \text{if } I \not\models F. \end{cases}$$

Now we can define the rule extraction problem formally. It is the problem of constructing a propositional representation describing the behaviour of some output unit of the given network.

Definition 6.1.4 (Rule Extraction Problem) *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network. The* rule extraction problem *for some $A \in \mathcal{U}_{\mathsf{out}}$ is the construction of a propositional logic formula F such that F is a propositional representation over $\mathcal{U}_{\mathsf{inp}}$ for A.*

Usually not all possible input combinations make sense in a given application domain, because they would correspond to invalid states of the world. Only a (small) subset of all combinations is allowed and we refer to it as *valid inputs*. Even more important here is the fact that all training samples are taken from this subset. Therefore, the network learns to solve a task under the implicit conditions hidden in the selection of the input. Integrity constraints are a way to make those conditions explicit during the extraction process. In fact, an integrity constraint can be an arbitrary formula describing the valid input combinations:

Definition 6.1.5 (Integrity Constraint, IC) *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network and let $V \subseteq \mathcal{P}(\mathcal{U}_{\mathsf{inp}})$ be the set of valid input combinations. A propositional formula* IC *is call* integrity constraint (IC) *for $\mathcal{N}$ and V, if $v \models \mathrm{IC}$ for all $v \in V$ and $w \not\models \mathrm{IC}$ for all $w \subseteq \mathcal{U}_{\mathsf{inp}}$ and $w \notin V$.*

Using the notion of integrity constraint, we can redefine the rule extraction problem appropriately. Because we are only interested in valid inputs, we restrict the set of interpretations from Definition 6.1.4 to those satisfying the given integrity constraint.

Definition 6.1.6 (Rule Extraction Problem wrt. Integrity Constraints) *Let* IC *be an integrity constraint for the threshold feed-forward network $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$. The* rule extraction problem *for some $A \in \mathcal{U}_{\mathsf{out}}$ is the construction of a propositional logic formula F over $\mathcal{U}_{\mathsf{inp}}$ such that for all interpretations I with $I \models \mathrm{IC}$ we find*

$$f_o(A, \iota_{\mathcal{N}}(I)) = \begin{cases} t^+ & \text{if } I \models F \\ t^- & \text{if } I \not\models F. \end{cases}$$

And for all interpretations J with $J \not\models \mathrm{IC}$ we find $J \not\models F$. We call F a propositional representation for A wrt. IC.

Lemma 6.1.7 *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network, let* IC *be an integrity constraint over $\mathcal{U}_{\mathsf{inp}}$. Let F be a propositional representation for some $A \in \mathcal{U}_{\mathsf{out}}$ and let G be a propositional representation for A wrt.* IC*. Then we find $G \equiv F \wedge \mathrm{IC}$.*

> **Proof** For $I \models \mathrm{IC}$ the we find $G^I = F^I$ by definition. For $J \not\models \mathrm{IC}$ we find $J \not\models G$ by Definition 6.1.6 and, hence, $G \equiv F \wedge \mathrm{IC}$. □

In the following sections, we introduce a novel approach to the rule extraction problem. It is a decompositional approach which employs binary decision diagrams as intermediate representation. This results in a very compact representation and leads to a fast algorithm.

6.2 CoOp – A New Decompositional Approach

In this section, we present a new decompositional approach for the extraction of knowledge from acyclic (feed-forward) neural networks. We first concentrate on threshold units and discuss how to apply the approach to smooth activation functions afterwards. The approach is presented in six steps:

1. *Decomposition:* A given network is split into its basic functional units, namely perceptrons (a single unit together with its incoming connections).

2. *Extraction of coalitions and oppositions from a given perceptron:* We show how to compute the sets of minimal coalitions and oppositions from a given perceptron. Those can be thought of as a set-based description of necessary conditions for a perceptron to be active and inactive, respectively. For this purpose, we develop a search tree together with some suitable pruning rules. Then we show how to extract binary decision diagrams (BDDs), which allow a much more compact representation of the results.

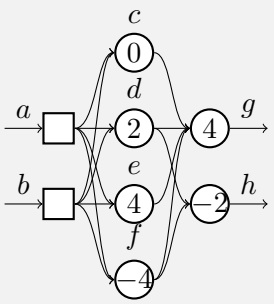
3. *Composition of the Results:* We show how to compose the results and in particular the resulting BDDs.

4. *Incorporation of Integrity Constraints:* Using integrity constraints, we can minimise the resulting diagrams. In contrast to existing approaches, we are able to use the integrity constraints throughout the whole extraction process and not only after compiling the final result.

5. *From Binary Decision Diagrams to Propositional Logic Programs:* By transforming the reduced ordered binary decision diagrams into propositional logic programs, we obtain a method to extract logic programs from acyclic neural networks, such that the T_P-operator of the extracted program mimics the input-output-behaviour of the network exactly.

A small feed-forward network of threshold units is shown in Figure 6.2.1. It serves as a running example throughout this section.

Example 6.2.1 A small feed-forward network of threshold units, which serves as running example throughout this section. The weights are shown in the table on the right and the numbers within the units denote the thresholds. a and b are the input and g and h are the output units for this networks.



ω	**c**	**d**	**e**	**f**	**g**	**h**
a	1	-2	5	2		
b	1	1	-3	1		
c					1	0
d					2	-3
e					3	0
f					5	-2

6.3 Decomposition of Feed-Forward Networks

We decompose the network into simple perceptrons, i.e., a single unit together with its incoming connections. In the following sections we show how to extract rules from those perceptrons. Here, we introduce the required notations, namely *perceptron*, *input pattern*, *coalition* and *opposition*.

Definition 6.3.1 (Perceptron) *Let $\theta, t^-, t^+ \in \mathbb{R}$ with $t^- < t^+$, $\mathcal{I}$ be a finite set of input symbols and $\omega : \mathcal{I} \to \mathbb{R}$. Then, $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ is called a* perceptron *and the corresponding function $f_\mathcal{P}$ is defined as follows:*

$$f_\mathcal{P} : \mathcal{P}(\mathcal{I}) \to \{t^-, t^+\}$$
$$I \mapsto \Theta_{t^-}^{t^+}\left(t^+ \cdot \sum_{A \in I} \omega(A) + t^- \cdot \sum_{B \in \mathcal{I} \setminus I} \omega(B) - \theta\right)$$

We call a perceptron positive *if all weights are above or equal to 0, and we call it* negative *if all weights are below or equal to 0.*

The function $f_\mathcal{P}$ for a given perceptron $\mathcal{P}$ is defined over subsets of the input symbols. All inputs contained in the set are assumed to be active and the others are inactive. For all active inputs we use the value t^+ and for the inactive ones t^-. The set of perceptrons for a given threshold network $\mathcal{N}$ is computed by collecting all non-input units together with their incoming weights.

Definition 6.3.2 (Set of Perceptrons) *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network. The set* $\mathrm{per}(N)$ *of all contained perceptrons is defined as follows:*

$$\mathrm{per}(N) := \left\{ \left(A, \left\langle t^-, t^+, \theta_A, \mathcal{I}, \omega_A \right\rangle\right) \mid A \in \mathcal{U} \setminus \mathcal{U}_{\mathsf{inp}}, \theta_A = \theta(A), \mathcal{I} = \{U \mid (U, A) \in \mathcal{C}\} \text{ and } \omega_A : I \mapsto \omega(I, A) \right\}$$

Notation 6.3.3 *We will use $\mathcal{P}_A$ to denote the unit A understood as perceptron, i.e., together with its incoming connections.*

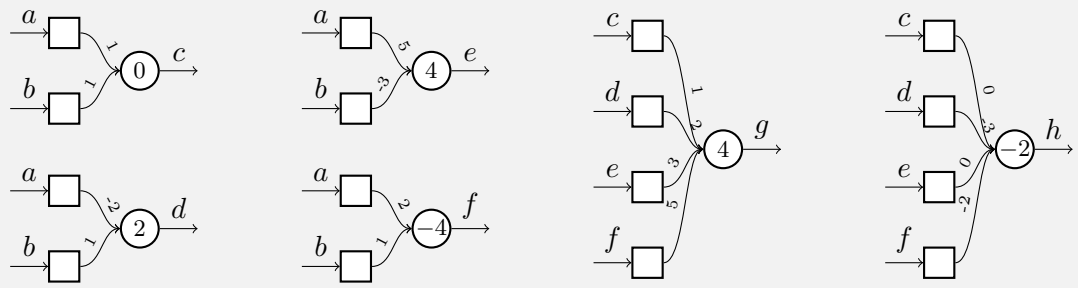
A Prolog implementation to compute the set of perceptrons for a given feed-forward network is shown in Figure 6.1. The resulting perceptrons after decomposing the network from above are shown in Example 6.3.1.

```
1 tffn2perceptrons(tffn(U, Ui, _Uo, C, Tn:Tp), Pers) :-
2     Pers := { A=per(Tn:Tp,T,IW) | unit(A,T) in (U subtract Ui),
3                                   IW := { I:W | I/A/W in C } }.
```

Figure 6.1: *Implementation to compute the set of perceptrons for a given feed-forward network: For each unit* `unit(A,T)`*, the set* `IW` *of incoming connections is composed and the resulting perceptron is added as* `A = per(Tn : Tp, T, IW)`*.*

In the sequel, we fix some of the input units U of a given perceptron to be active. This is done by *input patterns*. All units occurring in an input pattern $I \subseteq U$ are considered to be active while the state of the non-included input units is not fixed. I.e., some of the remaining units in $U \setminus I$ might be active as well. Therefore, an input pattern defines an upper and lower bound on the possible input of the perceptron.

Example 6.3.1 The perceptrons obtained by decomposing the network from Figure 6.2.1:



Definition 6.3.4 (Input Pattern) *Let $\mathcal{P}_A = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a perceptron. A subset $I \subseteq \mathcal{I}$ is called an* input pattern. *The minimal and maximal input wrt. the input pattern I are defined as:*

$$i_{\min}(I) = t^+ \cdot \sum_{A \in I} \omega(A) + \sum_{A \in \mathcal{I} \setminus I} \min(t^- \cdot \omega(A), t^+ \cdot \omega(A)) \quad \textit{and}$$
$$i_{\max}(I) = t^+ \cdot \sum_{A \in I} \omega(A) + \sum_{A \in \mathcal{I} \setminus I} \max(t^- \cdot \omega(A), t^+ \cdot \omega(A)).$$

Example 6.3.2 For $\mathcal{P}_g$ from Example 6.3.1, $I = \{c, d\}$, $t^- = -1$ and $t^+ = 1$ we find

$$i_{\min}(I) = (1.0 + 2.0) + (-3.0 - 5.0) = -5.0 \quad \text{and}$$
$$i_{\max}(I) = (1.0 + 2.0) + (3.0 + 5.0) = 11.0.$$

An input pattern which turns a perceptron active (regardless of the activation of the non-included inputs), is called *coalition*. One that turns it inactive is called *opposition*.

Definition 6.3.5 (Coalition) *Let $\mathcal{P}_A$ be a perceptron with threshold θ, let I be some input pattern for $\mathcal{P}_A$ and let $i_{\min}(I)$ be the corresponding minimal input. I is called a* coalition, *if $i_{\min}(I) \geq \theta$. A coalition I is called* minimal, *if none of its subset $I' \subset I$ is a coalition. We use $\mathcal{C}_A$ to denote the set of minimal coalitions for perceptron $\mathcal{P}_A$.*

Example 6.3.3 For $\mathcal{P}_g$ from Example 6.3.1, we find that $I_1 = \{c, d, f\}$ and $I_2 = \{e, f\}$ are minimal coalition, because $i_{\min}(I_1) = (1.0 + 2.0 + 5.0) - (3.0) = 5.0 > 4.0$ and $i_{\min}(I_2) = (3.0 + 5.0) - (3.0) = 5.0 > 4.0$, and none of its subset is a coalition itself. $\{d, e, f\}$, $\{c, e, f\}$ and $\{c, d, e, f\}$ are also coalitions but not minimal due to the fact that they are supersets of I_2.

Definition 6.3.6 (Opposition) *Let $\mathcal{P}_A$ be a perceptron with threshold θ, let I be some input pattern for $\mathcal{P}_A$ and let $i_{\max}(I)$ be the corresponding maximal input. I is called an* opposition, *if $i_{\max}(I) < \theta$. An opposition I is called* minimal, *if none of its subset $I' \subset I$ is an opposition. We use $\mathcal{O}_A$ to denote the set of minimal oppositions for perceptron $\mathcal{P}_A$.*

Example 6.3.4 For $\mathcal{P}_e$ from Example 6.3.1, we find that $J = \{b\}$ is an opposition, as $i_{\max}(J) = (-3.0) + (5.0) = 2.0 < 4.0$.

Please note that neither coalitions nor oppositions need to exists for a given perceptron. E.g., for the perceptron $\mathcal{P}_g$ from Example 6.3.1, there is no opposition, because we cannot fix inputs to be off for the moment. And, there can be multiple minimal coalitions and minimal oppositions. But the (always existing) set of coalitions and the set of oppositions can be used to fully describe the behaviour of a perceptron. Furthermore, it suffices to consider the sets of minimal coalitions and oppositions, which are uniquely determined.

6.4 Computing Minimal Coalitions and Oppositions

Here, we show how to construct the set of minimal coalitions for a given positive perceptron and the set of minimal oppositions from a negative perceptron. As both algorithms are very similar, we present the extraction of coalitions in detail and mention the differences afterwards. The final algorithm is based on binary decision diagrams as introduced in Section 2.4. But to explain the underlying intuitions, we construct a tree to guide the search. This tree is pruned to obtain a faster algorithm. Afterwards, we use some symmetry in those trees to construct BDDs. This yields an algorithm that extracts the BDD directly from the perceptron.

6.4.1 A Search Tree to Find Coalitions

We use a search tree to guide the extraction of coalitions from a given perceptron. The nodes of the tree are labelled with input patterns as defined above and the corresponding minimal input. Starting with the root node which is labelled with the empty input pattern, all children are obtained by adding one input symbol. Furthermore, the children are sorted descending wrt. their minimal input.

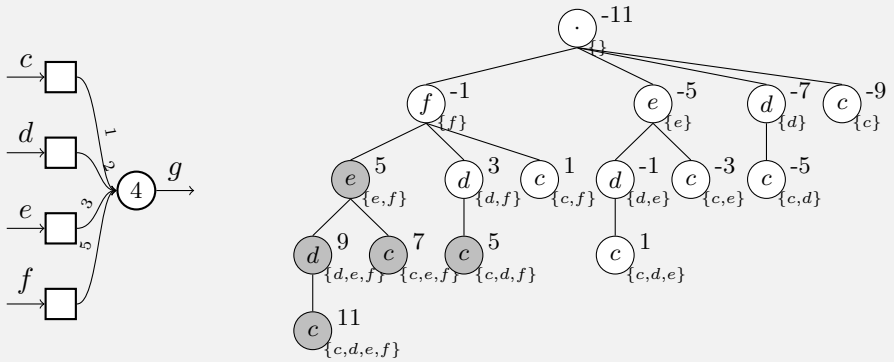
Definition 6.4.1 (Search Tree Node) *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a positive perceptron. Let $\prec$ be a fixed linear order on $\mathcal{I}$ such that $A \prec B$ if $\omega(A) \geq \omega(B)$. A search tree node is a pair $\langle I, C \rangle$, with $I \subseteq \mathcal{I}$ and C being a list of child nodes. For each child node $\langle I_i, C_i \rangle$ we find $I_i = I \cup \{A_i\}$ with $A \prec A_i$ for all $A \in I$. The list of children is sorted descending with respect to the minimal input of the corresponding I_i.*

Next, we define the *full coalition search tree*. This tree consists of search tree nodes which are maximal expanded in the sense that all nodes contain all possible child nodes. E.g., the root node for a given perceptron with n inputs has n child nodes.

Definition 6.4.2 (Full Coalition Search Tree) *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a positive perceptron. The* full coalition search tree $\mathsf{ST}_{\mathcal{P}}$ *for $\mathcal{P}$ is defined to be the search tree node $\langle \emptyset, C \rangle$ such that for all nodes $\langle I', C' \rangle$ in the tree, we find $|C'| = |\{\mathcal{I}\}| - |I'|$.*

The search tree corresponding to perceptron $\mathcal{P}_g$ from above is shown in Example 6.4.1. Please note that it actually suffices to store the added input symbol B for each node, because the resulting input pattern can be obtained by following the path to the root node. In the sequel we show two important properties of those trees, namely completeness and sortedness wrt. the minimal input.

Example 6.4.1 The perceptron $\mathcal{P}_g$ from Example 6.3.1 is shown on the left and the resulting full coalition search tree on the right. For each node $\langle I, C\rangle$ in the tree we show a circle containing the symbol A that was added to the parents I or a "$\cdot$" for the root node. All nodes are annotated with I and $i_{\min}(I)$ and those for which I is a coalition are shown with gray background.



The corresponding set of coalitions is $\{\{e, f\}, \{d, e, f\}, \{c, d, e, f\}, \{c, e, f\}, \{c, d, f\}\}$ and the minimal coalitions are $\mathcal{C}_g = \{\{e, f\}, \{c, d, f\}\}$.

Lemma 6.4.3 *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega\rangle$ be a positive perceptron and let $\mathsf{ST}_\mathcal{P}$ be the corresponding full coalition search tree. Then $\mathsf{ST}_\mathcal{P}$ is* complete wrt. *$\mathcal{P}$, i.e., for each input pattern $I \subseteq \mathcal{I}$ there exists a node $\langle I, C\rangle$ in $\mathsf{ST}_\mathcal{P}$.*

> **Proof (sketch)** This can easily be shown by induction on the size s of the input pattern. For $s = 0$, the resulting input pattern $\{\}$ is contained in the tree, namely as root node. Under the assumption that all input patterns of size $n < |\mathcal{I}|$ are contained, we can conclude that all of size $n + 1$ are contained, because each node is maximally expanded, i.e., for each direct superset there is a child node. □

Lemma 6.4.4 *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega\rangle$ be a positive perceptron and let $\mathsf{ST}_\mathcal{P}$ be the corresponding coalition search tree. Then, the minimal input of a child node is greater or equal than the minimal input of the parent and it is greater or equal than the minimal input of all right siblings.*

> **Proof (sketch)** The minimal input of some node is greater or equal than the minimal input of the parent node, because we consider positive inputs only and by definition we find that the corresponding input pattern of the parent is a subset of the childs input pattern. Therefore, more inputs are fixed to be active which ensures a greater or equal minimal input. By definition all right siblings have smaller or equal input, because the list of children is sorted. □

A Prolog implementation to construct a coalition search tree is shown in Figure 6.2. This implementation follows Definition 6.4.2 by fully expanding each node recursively. We can compute the set of coalitions for a given positive perceptron by traversing the corresponding search tree and collecting all those input patterns for which the minimal input is above the threshold. Because the tree is complete in the sense introduced in Lemma 6.4.3, we can conclude that all coalitions can be found this way. But the sortedness shown in Lemma 6.4.4 gives rise to pruning methods which is discussed below.

```
coSearchtree(Name=per(Tn:Tp, Threshold, IWs), Node) :-
    % the perceptron must be positive
    forall( (_:W) in IWs, W>=0),
    %PInputs := maplist(posWeight, IWs),
    % sort the inputs according to the weights
    SIWs := sort(IWs, weightCompare),
    % compute the minimal input wrt.  the empty input pattern
    Inp := Tn*sum({{ W | (_:W) in SIWs }}), !,
    coChildren(Tn:Tp, Name, [], SIWs, Threshold, Inp, Children),
    Node = node(Name, [], Inp, Children).

% if there are no inputs left, then the list of children is empty
coChildren(_, _Prefix, _PIP, [], _Threshold, _, []).

% construct a node for the first element of the inputs
coChildren(Tn:Tp, Prefix, PIP, [I:W|WIs], Threshold, Inp, [Node|Siblings]) :-
    Inp2 := Inp - Tn*W + Tp*W,
    coChildren(Tn:Tp, I:Prefix, [I|PIP], WIs, Threshold, Inp2, Children),
    Node = node(I:Prefix, [I|PIP], Inp2, Children),
    coChildren(Tn:Tp, Prefix, PIP, WIs, Threshold, Inp, Siblings).
```

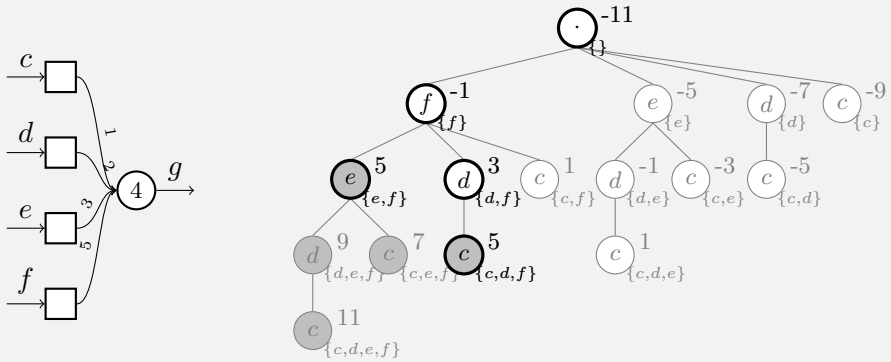
Figure 6.2: *Implementation to construct a search tree to guide the extraction process. First, we check whether the perceptron is positive. The inputs are sorted wrt. the weights. Then, the minimal input wrt. the empty input pattern is computed and all children of the root node are computed, which recursively computes all sub-children.*

6.4.2 A Pruned Search Tree to Find Coalitions

Obviously, some of the nodes in the full coalition search tree are irrelevant wrt. the set of minimal coalitions. Namely all nodes representing non-minimal coalitions and all nodes below which there is no coalition at all. Lemma 6.4.4 tells us that the search tree is sorted with respect to the minimal inputs, i.e., the minimal input of a child node is greater or equal than the minimal input of the parent and it is greater or equal than the minimal input of all left siblings. Therefore, we do not need to look at the child nodes of coalitions because they are coalitions as well. And, if we find a node that is not a coalition and neither is one of its children, we can prune it and all right siblings. This is due to the fact that their minimal input is even smaller and thus they cannot be coalitions either. Using those two ideas for pruning, we derive the following notion of a *pruned coalition search tree*. The question remains how to find those nodes under which there are no coalitions. This can be done by computing the maximal increase of the minimal input for a node. If this added to the minimal input does not exceed the threshold, we can conclude that the node is irrelevant. The maximal increase can be computed by adding the weights of all remaining input symbols and multiplying it by $(t^+ - t^-)$.

Definition 6.4.5 (Irrelevant Node, Pruned Coalition Search Tree) *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a positive perceptron and let $\mathsf{ST}_\mathcal{P}$ be the corresponding coalition search tree as defined above for the linear order $\prec$. Let $i(I)$ be the maximal element of some $I \subseteq \mathcal{I}$ wrt. $\prec$ and let $J(I) \subseteq \mathcal{I}$ be the set of all inputs which are bigger than $i(I)$, i.e., $J(I) := \{j \mid j \in \mathcal{I} \text{ and } i(I) \prec j\}$. Let $r(I) = (t^+ - t^-) \cdot \sum_{j \in J(I)} \omega(j)$. A node $\langle I, C \rangle$ in $\mathsf{ST}_\mathcal{P}$ is called* irrelevant *if I is a non-minimal coalition, or if $i_{\min}(I) + r(I) < \theta$. The* pruned coalition search tree *$\mathsf{PT}_\mathcal{P}$ is obtained from $\mathsf{ST}_\mathcal{P}$ by removing all irrelevant nodes.*

Example 6.4.2 The perceptron $\mathcal{P}_g$ from Example 6.3.1 and the resulting search trees. The full tree is shown in gray and the pruned tree in black on top of it. As before, coalitions are shown with gray background and the annotations next to the nodes show the corresponding input pattern and the minimal input.



The corresponding set of minimal coalitions is $\mathcal{C}_g = \{\{e, f\}, \{c, d, f\}\}$.

Please note that if all nodes are irrelevant, the pruned search tree is empty. Example 6.4.2 shows the perceptron $\mathcal{P}_g$ from Example 6.3.1, the resulting full search tree and overlayed, the pruned search tree. The following lemma states that all minimal coalitions are still nodes contained in the pruned search tree.

Lemma 6.4.6 *Let $\mathcal{P}$ be a perceptron, $\mathcal{C}_\mathcal{P}$ be the set of minimal coalitions and let $\mathsf{PT}_\mathcal{P}$ be the corresponding pruned coalition search tree. Then we find that for each $c \in \mathcal{C}_\mathcal{P}$ there is a node $\langle I, C\rangle$ in $\mathsf{PT}_\mathcal{P}$ with $I = c$.*

> **Proof** By definition, we only remove irrelevant nodes, which are defined to be either non-minimal coalition nodes, to be leaf nodes which are not coalitions or to be nodes such that all children are irrelevant. But none of those can be a node corresponding to a minimal coalition. Hence, the pruned search tree does still contain all minimal coalition nodes. □

Corollary 6.4.7 *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega\rangle$ be a perceptron and let $\mathsf{PT}_\mathcal{P}$ be the corresponding pruned coalition search tree. Then, all leaf nodes $\langle I, []\rangle$ in $\mathsf{PT}_\mathcal{P}$ correspond to minimal coalitions.*

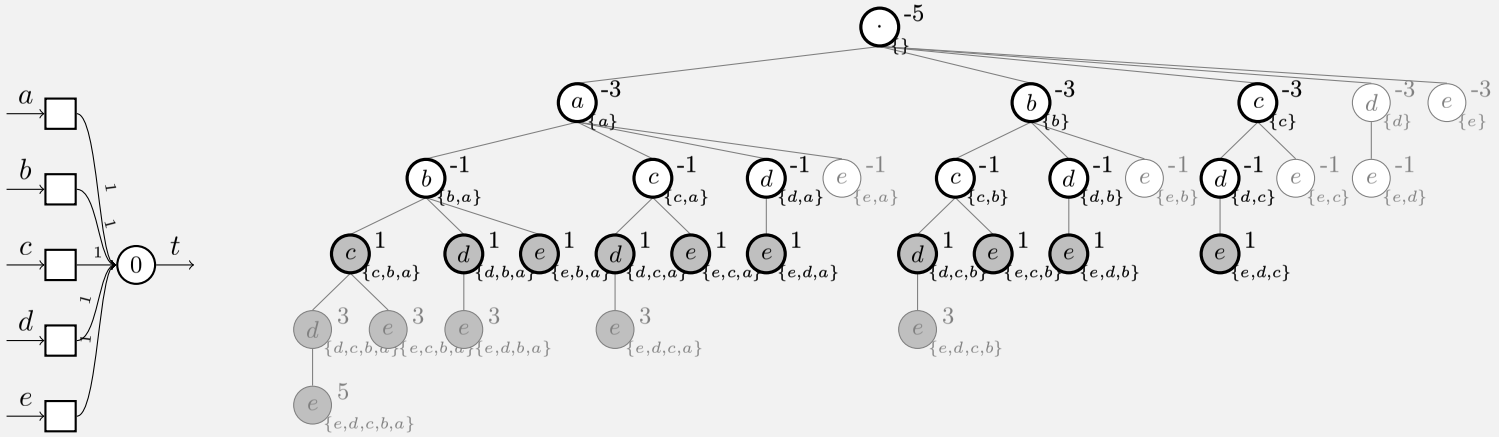
Figure 6.3 shows a Prolog implementation to construct the pruned search tree. The two pruning steps are captured in the first and second clause of `coPChildren/8`. If the minimal input exceeds the threshold, we can prune all child nodes and we know that we found a coalition, this is done by setting the last argument to []. The seventh argument is always instantiated with $r(I)$ from Definition 6.4.5. Therefore, we can prune the tree as done in the second clause. If neither pruning condition applies, we expand the node (third clause) by computing its children and the right siblings.

Even though the pruning methods improve the search, there are search trees, where further pruning can be done. Example 6.4.3 presents a slightly larger perceptron with a search tree showing certain symmetries. Those symmetries are used to convert the pruned search tree into an *equivalent* binary decision diagram.

```
1  coPSearchtree(Name=per(Tn:Tp, Threshold, IWs), Node) :-
2      % the perceptron must be positive
3      forall( (_:W) in IWs, W>=0),
4      % sort the inputs according to the weights
5      SIWs := sort(IWs, weightCompare),
6      % compute the minimal input wrt.  the empty input pattern
7      Inp := Tn*sum({{ W | (_:W) in SIWs }}), !,
8      % compute the maximal increase of input wrt.  the empty input pattern
9      RInp := (Tp-Tn)*sum({{ W | (_:W) in SIWs }}), !,
10     coPChildren(Tn:Tp, Name, [], SIWs, Threshold, Inp, RInp, Children),
11     Node = node(Name, [], Inp, Children).
12
13 % if the minimal input is above the threshold, then there are no children
14 coPChildren(_, _, _, _, T, Inp, _, []) :- Inp >= T, !.
15
16 % if the maximal increase of input does not suffice, prune the tree
17 coPChildren(_, _, _, _, T, Inp, RInp, []) :- Inp + RInp < T, !.
18
19 % construct a node for the first element of the inputs
20 coPChildren(Tn:Tp, Prefix, PIP, [I:W|WIs], T, Inp, RInp, [Node|Siblings]) :-
21     Inp2 is Inp - Tn*W + Tp*W,
22     RInp2 is RInp - (Tp-Tn)*W,
23     coPChildren(Tn:Tp, I:Prefix, [I|PIP], WIs, T, Inp2, RInp2, Children),
24     Node = node(I:Prefix, [I|PIP], Inp2, Children),
25     coPChildren(Tn:Tp, Prefix, PIP, WIs, T, Inp, RInp2, Siblings).
```

Figure 6.3: *Implementation to construct the pruned search tree: As before, we check whether the perceptron is positive and sort the inputs. Then the root node is expanded which recursively constructs all sub-nodes. The implementation follows Definition 6.4.5 by pruning all children of coalition nodes (first clause for* `coPChildren/8`*) and pruning a node and its right siblings if the maximal increase of the minimal input is not sufficient to exceed the threshold.*

Example 6.4.3 A perceptron encoding a 3-of-5 decision. If 3 of the 5 input units are active, the output unit is activated as well. The corresponding (pruned) search tree:



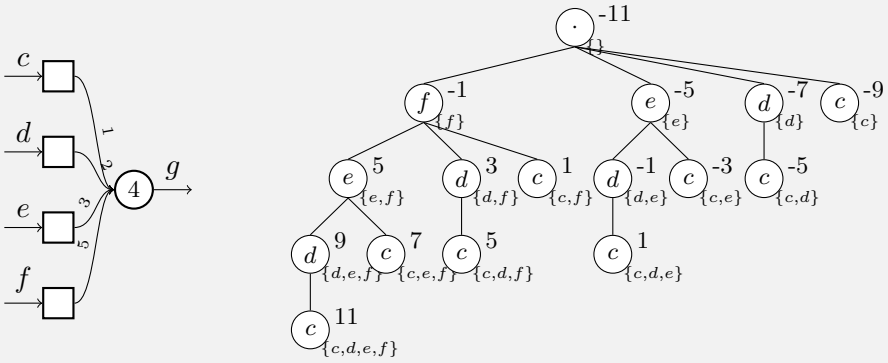
The set of minimal coalitions is $\mathcal{C}_t = \{\{a,b,c\},\{a,b,d\},\{a,b,e\},\{a,c,d\},\{a,c,e\},\{a,d,e\},\{b,c,d\},\{b,c,e\},\{b,d,e\},\{c,d,e\}\}$.

6.4.3 Reduced Ordered Binary Decision Diagram Representing Coalitions

Here we show how to convert the pruned search tree into an ordered binary decision diagram. Looking at the tree shown in Example 6.4.1 and ignoring the colours, we find the following relations between neighbouring siblings:

- Let l be the direct left brother of r and let l_l be the leftmost sub tree of l. Then the minimal inputs of r can be obtain from l_l by adding $(t^- - t^+) \cdot w_l$ with w_l being the weight corresponding to l.
- Let l_c be the list of sub-trees of l without l_l, and let r_c be the list of all sub-trees of r. Then the minimal inputs of some sub-tree in r_c can be obtain from those of l_c by adding $(t^- - t^+) \cdot w_l + (t^+ - t^-) \cdot w_r$.

Example 6.4.4 To show the symmetries in the search tree, we omitted the background colours indicating coalitions:



We use the corresponding I's to reference a node in the tree which is used to exemplify the observations from above.

- Let $\{f\}$ be l and $\{e\}$ be r, then we find l_l to be $\{e, f\}$ and $w_l = 5$. The input patterns of the nodes below r can be obtained by removing f from the input patterns below l_l. Thus, the minimal inputs of all nodes below r, can be computed by adding $(t^- - t^+) \cdot w_l = (-1 - 1) \cdot 5 = -10$ to the minimal inputs of l_l.
- We find furthermore l_c to be $[\{d, f\}, \{c, f\}]$ and r_c to be $[\{d, e\}, \{c, e\}]$. The minimal inputs of the trees in r_a can be obtained from those in l_r by adding $(t^- - t^+) \cdot w_l + (t^+ - t^-) \cdot w_r = (-1 - 1) \cdot 5 + (1 - -1) \cdot 3 = -4$. The corresponding input patterns can be obtained by replacing f by e.

Those observations, again shown in Example 6.4.4, are the basis of the constructions presented below. In particular the second observation, helps to omit the expansion of the right sibling under certain circumstances. Let the sub-tree l contain a coalition and let d_l denote the minimal distance between the minimal input of a coalition below l and the threshold. Then d_l can be understood as a safety margin. If $(t^- - t^+) \cdot w_l + (t^+ - t^-) \cdot w_r \leq d$, then, a node in r_c is a coalition if and only if the corresponding node in l_c is. I.e., if we find a way to "link" r and l_c, we do not need to expand r. Unfortunately, those links cannot be represented in trees. Therefore, we construct reduced ordered binary decision diagrams as introduced in Section 2.4. First, we give a formal definition, then we discuss properties of those BDDs and afterwards present a construction based on the search trees introduced above.

Notation 6.4.8 *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega\rangle$ be a perceptron and let $\prec$ be a linear order on $\mathcal{I}$ such that $A \prec B$ if $\omega(A) \geq \omega(B)$. Let $n = \langle I, C\rangle$ be a search tree node, then we use:*

- $\mathrm{id}(n)$ *as a unique identifier, as e.g., the corresponding input pattern.*
- $\mathrm{var}(n)$ *to denote the maximal element of C wrt. $\prec$, i.e., the input symbol that was added.*
- $\mathrm{rs}(n)$ *to denote the right sibling of n, if existent.*
- $w(n) := \omega(\mathrm{var}(n))$.
- $\mathrm{mci}(n)$ *to denote the minimum of the minimal inputs of all leaf nodes below n.*

Definition 6.4.9 (Coalition BDD) *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega\rangle$ be a positive perceptron with $0, 1 \notin \mathcal{I}$, let $\mathsf{PT}_\mathcal{P}$ be the corresponding pruned search tree and let $\prec$ be a linear order on $\mathcal{I}$ such that $A \prec B$ if $\omega(A) \geq \omega(B)$. We define the set N and R as follows:*

- *If $\mathsf{PT}_\mathcal{P}$ is empty, then $R = 0$ and $N = \{\}$.*
- *If $\mathsf{PT}_\mathcal{P}$ contains only the root node, then $R = 1$ and $N = \{\}$.*
- *Otherwise let $R = \mathrm{id}(r_l)$ for the leftmost child r_l of the root node and let N be the (initially empty) set of BDD nodes $\langle i, v, h, l\rangle$ defined as follows:*
  1. *For each leaf node n in $\mathsf{PT}_\mathcal{P}$, add the node $\langle \mathrm{id}(n), \mathrm{var}(n), 1, l\rangle$ to N with $l = 0$ if n has no right sibling or $l = \mathrm{id}(\mathrm{rs}(n))$.*
  2. *Let $n = \langle I, C\rangle$ be a tree node and let l be its left sibling with $\langle \mathrm{id}(l), \mathrm{var}(l), h_l, l_l\rangle$ being the BDD node constructed for l. And let $\langle l_l, \mathrm{var}(n), h_{l_1}, l_{l_1}\rangle$ be the BDD node corresponding to the leftmost child of l. Let $d_l = \mathrm{mci}(l) - \theta$ and let $d = d_l + (t^- - t^+) \cdot w(l) + (t^+ - t^-) \cdot w(n)$. If $d > 0$ add $\langle \mathrm{id}(n), \mathrm{var}(n), l_{l_1}, l_n\rangle$ to N with $l_n = 0$ if n has no right sibling or $l_n = \mathrm{id}(\mathrm{rs}(n))$.*
  3. *For each non-root tree node $n = \langle I, C\rangle$ in $\mathsf{PT}_\mathcal{P}$ for which none of the other cases applies, add a node $\langle \mathrm{id}(n), \mathrm{var}(n), \mathrm{id}(c), l\rangle$ to N with c being the left-most child of n and $l = 0$ if n has no right sibling or $l = \mathrm{id}(\mathrm{rs}(n))$.*

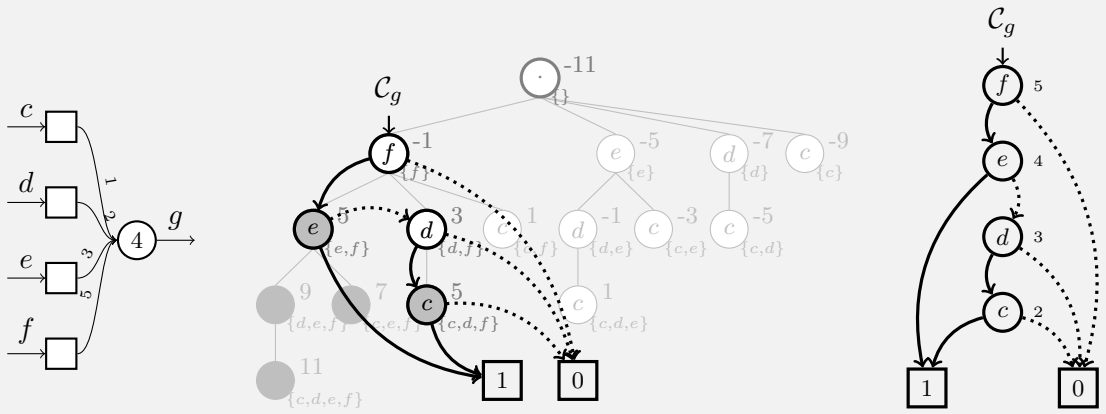
The coalition ROBDD for $\mathcal{P}$ with leaf nodes 0 and 1 and root node R *is the reduced ordered binary decision diagram* $\mathrm{coBDD}_\mathcal{P}$ *with* $\mathrm{coBDD}_\mathcal{P} := \mathrm{robdd}(\prec, 0, 1, R, N)$.

Example 6.4.5 shows the coalition ROBDD for our running example. It shows the search tree from above, the BDD drawn on top of it as well as the final coalition BDD on the right.

Lemma 6.4.10 *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega\rangle$ be a perceptron and let $\prec$ be a linear order on $\mathcal{I}$ such that $A \prec B$ if $\omega(A) \geq \omega(B)$. Let R and N be constructed wrt. $\prec$ as in Definition 6.4.9. Then the BDD $\langle 0, 1, R, N\rangle$ is ordered wrt. $\prec$.*

Proof We show that all nodes added to N are ordered wrt. $\prec$. Following the construction of the BDD, we find that a node links usually to nodes which are below it, or are right siblings. In both situations, we find that the corresponding variables are larger wrt. $\prec$. There is only one exception introduced by step 2. in Definition 6.4.9. There, the high branch of some node points to a node in the tree below its left sibling. But the node it links to is the low branch of the leftmost child of the left sibling. Looking at the underlying search tree, we find that the added input symbol of the leftmost child of the left sibling coincides with the added input symbol of the current node. Therefore, we find that the variable of its low branch is larger wrt. $\prec$, which ends the proof. □

Example 6.4.5 The perceptron $\mathcal{P}_g$ from Example 6.3.1 is shown on the left. The search tree from above and overlayed the resulting BDD is depicted in the middle and the pure BDD on the right.



The corresponding set of minimal coalitions is $\mathcal{C}_g = \{\{e, f\}, \{c, d, f\}\}$ and the paths from root to 1 are $[5, 4, 1]$ and $[5, 4, 3, 2, 1]$, corresponding to $\{e, f\}$ and $\{c, d, f\}$, respectively. The pure BDD on the right is annotated with the node ids used in the paths.

To construct the sets of minimal coalitions, we need to collect all satisfying assignments for the corresponding BDD. A satisfying assignment can be constructed by following a path from the root to 1. If this path follows a high branch for a given node, the corresponding variable must be mapped to true in the assignment and to false otherwise. In our particular case, we only need to collect those variables which are mapped to true in the assignment, because whenever there exists a path following a low branch, then there also is a path to 1 which follows the high branch. This results from the construction of the BDD and the fact that all weights are positive.

Lemma 6.4.11 *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a positive perceptron, let $\mathcal{C}_\mathcal{P}$ be the set of minimal coalitions and let* $\text{coBDD}_\mathcal{P} = \langle 0, 1, R, N \rangle$ *be the corresponding coalition ROBDD. Then, for each coalition $c \in \mathcal{C}_\mathcal{P}$, there exists a path $[n_1, \ldots, n_c]$ from the root $n_1 = R$ to $n_c = 1$ in* $\text{coBDD}_\mathcal{P}$ *such that $c = \{\text{var}(n_i) \mid n_{i+1} = \text{high}(n_i)\}$. And for each path $[n_1, \ldots, n_c]$ from the root $R = n_1$ to $1 = n_c$ in* $\text{coBDD}_\mathcal{P}$ *there exists a coalition $c \in \mathcal{C}_\mathcal{P}$ such that $c = \{\text{var}(n_i) \mid n_{i+1} = \text{high}(n_i)\}$.*

Proof (sketch) This lemma follows directly from Lemma 6.4.6, Corollary 6.4.7 and the construction of the BDD. Corollary 6.4.7 tells us that every leaf node of the underlying pruned search tree corresponds to a coalition. For all those nodes there are links to 1 in the BDD. Furthermore, we find for each branch in the tree a path in the BDD visiting the corresponding nodes. □

Lemma 6.4.12 *Let $\mathcal{P}$ be a perceptron. Then* $\text{coBDD}_\mathcal{P}$ *is uniquely defined for each order $\prec$.*

Proof This follows directly from the uniqueness of ROBDDs and Lemma 6.4.11. □

We can now construct a coalition ROBDD for a given perceptron using ideas from the pruned search tree constructed above. The construction is based on a left depth first traversal of the search tree, starting from the root node. While traversing the tree, we collect the nodes of a BDD following Definition 6.4.9. I.e., for each node we check all four cases and construct the

corresponding BDD node. Figure 6.4 shows a Prolog implementation. The first two clauses for `coBDD/9` implement the pruning steps of the pruned search tree described above. The third clause implements the back-link to the left sibling and the fourth the usual left-depth first expansion.

According to Definition 6.4.9 we would have to reduce and order the BDD afterwards. Please note that we can do this already while collecting the nodes. As shown in Lemma 6.4.10, we know that the BDD is already ordered, i.e., we have to reduce it only. By calling `makeBDDNode/5` to create the node, only those nodes are actually created that are really necessary. If a node with the same variable, high and low branch already exists, that node is used instead of creating a new one. Example 6.4.6 shows the resulting BDD for the perceptron from Example 6.4.3. The BDD on the left is shown on top of the search tree and on the right purely.

6.4.4 Reduced Ordered Binary Decision Diagram Representing Oppositions

Here, we show how to construct opposition BDDs for negative perceptrons. As already mentioned, this algorithm is very similar to the one described above. Sorting the inputs ascending wrt. their weights has the following effects:

- The sum over all weights multiplied by t^- is now the maximal input wrt. the empty input pattern.
- The underlying search tree, is ordered exactly the other way. I.e., the maximal input of a child node is smaller or equal than the parent and it also smaller or equal than all right siblings.
- The value $r(I)$ introduced in Definition 6.4.5 does now represent the maximal decrease of the maximal input. Therefore, we can prune the tree if $i_{\max}(I) + r(I) \geq \theta$, because there are no opposition below the current node.

Figure 6.5 shows an implementation to construct the pruned search tree for coalitions and for oppositions. Both differ in the pre-processing and the pruning cases only. The pruned opposition tree for the (negative) perceptron $\mathcal{P}_h$ is shown in Example 6.4.7

As for the coalition BDDs, we base the construction of BDDs representing oppositions on the pruned search tree.

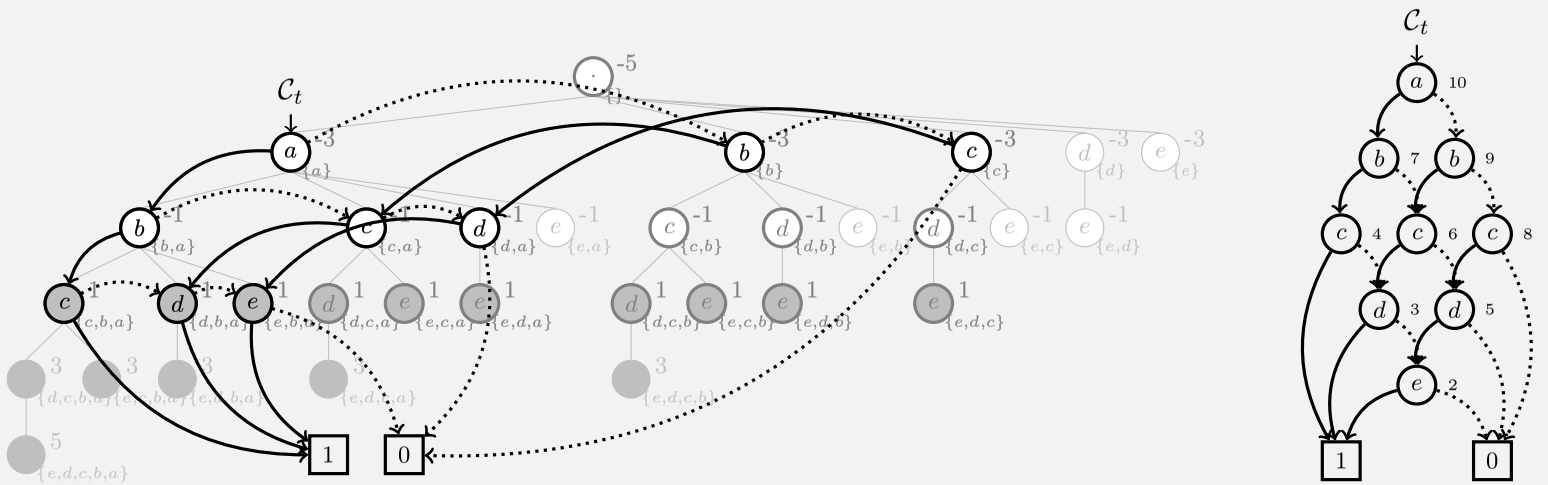
Definition 6.4.13 (Opposition BDD) *Let $\mathcal{P} = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a negative perceptron with $0, 1 \notin \mathcal{I}$ and let $\mathcal{O}_\mathcal{P}$ be the corresponding set of minimal oppositions. Let $\mathsf{PT}_\mathcal{P}$ be the corresponding pruned search tree and let $\prec$ be a linear order on $\mathcal{I}$ such that $A \prec B$ if $\omega(A) \leq \omega(B)$. We define the set N and R as follows:*

- *If $\mathsf{PT}_\mathcal{P}$ is empty, then $R = 0$ and $N = \{\}$.*
- *If $\mathsf{PT}_\mathcal{P}$ contains only the root node, then $R = 1$ and $N = \{\}$.*
- *Otherwise let $R = \mathrm{id}(r_l)$ for the leftmost child r_l of the root node and let N be the (initially empty) set of BDD nodes $\langle i, v, h, l \rangle$ defined as follows:*
  1. *For each leaf node n in $\mathsf{PT}_\mathcal{P}$, add the node $\langle \mathrm{id}(n), \mathrm{var}(n), 1, l \rangle$ to N with $l = 0$ if n has no right sibling or $l = \mathrm{id}(\mathrm{rs}(n))$.*

```
1  coBDD(Name=per(Tn:Tp, Th, IWs), B0:B1, R:BDD2) :-
2      % the perceptron must be positive
3      forall( (_:W) in IWs, W>=0),
4      % sort the inputs according to the weights
5      SIWs := sort(IWs, weightCompare),
6      VarOrder := {{ I | (I:_) in SIWs }},
7      % compute the minimal input wrt.  the empty input pattern
8      Inp := Tn*sum({{ W | (_:W) in SIWs }}), !,
9      RInp := (Tp-Tn)*sum({{ W | (_:W) in SIWs }}), !,
10     bddEmpty(VarOrder, B0, B1, BDD1),
11     coBDD(Tn:Tp:Th, Name, SIWs, Inp, RInp, nolb, BDD1, R:BDD2, _).
12
13 % If the minimal input exceeds the Th, this is equivalent to B1
14 coBDD(_:_:Th, _, _, Inp, _, _, BDD, B1:BDD, D) :-
15     Inp >= Th, !,
16     D is Inp - Th,
17     bddNode1(BDD, B1).
18
19 % if the maximal increse of the minimal input is not sufficient, prune
20 coBDD(_:_:Th, _, _, Inp, RInp, _, BDD, B0:BDD, nc) :-
21     Inp + RInp < Th, !,
22     bddNode0(BDD, B0).
23
24 % expand node wrt to some left neighbour if possible
25 coBDD(Tn:Tp:Th, Pref, [I:W|IWs], Inp, RInp, LBR/LBD/LBW, BDD, R:BDD2, D) :-
26     HD is LBD - Tp*LBW + Tn*LBW - Tn*W + Tp*W,
27     HD >= 0,
28     RInp2 is RInp - (Tp-Tn)*W,
29     bddNode(BDD, node(_, _, HR, LBR)),
30     coBDD(Tn:Tp:Th, Pref, IWs, Inp, RInp2, HR/HD/W, BDD, LR:LBDD, LD),
31     composeResults(LBDD, I:Pref, HR, HD, LR, LD, R:BDD2, D).
32
33 % expand node, first go down and then go right
34 coBDD(Tn:Tp:Th, Pref, [I:W|IWs], Inp, RInp, _, BDD, R:BDD2, D) :-
35     Inp2 is Inp - Tn*W + Tp*W,
36     RInp2 is RInp - (Tp-Tn)*W,
37     coBDD(Tn:Tp:Th, I:Pref, IWs, Inp2, RInp2, nolb, BDD, HR:HBDD, HD),
38     (   HD = nc % i.e., we can cut all right siblings
39     ->  ( bddNode0(BDD, R), BDD2=BDD, D=nc )
40     ;   ( coBDD(Tn:Tp:Th, Pref, IWs, Inp, RInp2, HR/HD/W, HBDD, LR:LBDD, LD),
41           composeResults(LBDD, I:Pref, HR, HD, LR, LD, R:BDD2, D) )).
42
43 composeResults(BDDin, V:ID, H, HD, L, nc, R:BDD, HD) :- !,
44     bddMakeNode(BDDin, node(V, H, L, V:ID), R, BDD).
45 composeResults(BDDin, V:ID, H, HD, L, LD, R:BDD, D) :-
46     D is min(HD, LD),
47     bddMakeNode(BDDin, node(V, H, L, V:ID), R, BDD).
```

Figure 6.4: *Implementation to construct the reduced ordered binary decision diagram representing the set of minimal coalition for a given perceptron $\mathcal{P}$: The first and second clause for* coBDD/9 *cover the pruning steps of the pruned search tree. The third clause establishes the link to the left sibling if possible and the fourth clause implements the left depth first expansion.*

Example 6.4.6 The pruned search tree from Example 6.4.3, overlayed with the corresponding coalition ROBDD:



The set of minimal coalitions together with the corresponding paths through the BDD:

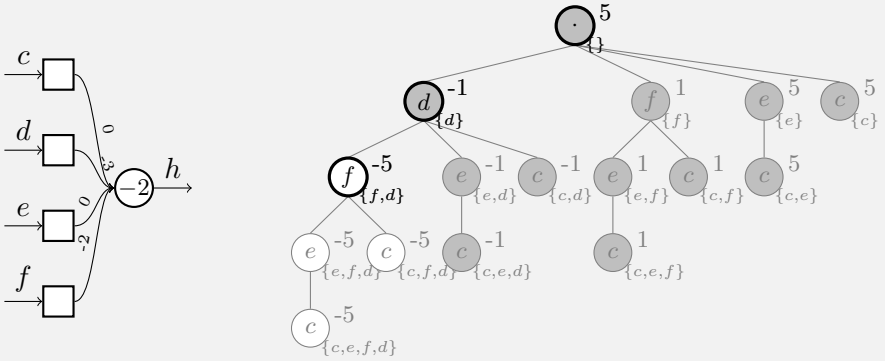
coalition	$\{a,b,c\}$	$\{a,b,d\}$	$\{a,b,e\}$	$\{a,c,d\}$	$\{a,c,e\}$
path	$[10,7,4,1]$	$[10,7,4,3,1]$	$[10,7,4,3,2,1]$	$[10,7,6,3,1]$	$[10,7,6,3,2,1]$
coalition	$\{a,d,e\}$	$\{b,c,d\}$	$\{b,c,e\}$	$\{b,d,e\}$	$\{c,d,e\}$
path	$[10,7,6,5,2,1]$	$[10,9,6,3,1]$	$[10,9,6,3,2,1]$	$[10,9,6,5,2,1]$	$[10,9,8,5,2,1]$

As you can see, only a small fraction of the nodes is actually visited while constructing the BDD. Only 9 from 32 nodes in the full and 22 from the pruned search tree have been visited.

```
coopPSearchtree(CO, Name=per(Tn:Tp, Threshold, IWs), Node) :-
    % the perceptron must be correct, i.e., positiv for co, negativ for op
    forall( (_:W) in IWs, correctWeight(CO, W) ),
    % sort the inputs according to the weights
    SIWs := sort(IWs, weightCompare(CO)),
    % compute the minimal/maximal input wrt.  the empty input pattern
    Inp := Tn*sum({{ W | (_:W) in SIWs }}), !,
    % compute the maximal impact of the remaining inputs for the empty input
    RInp := (Tp-Tn)*sum({{ W | (_:W) in SIWs }}), !,
    coopPChildren(CO, Tn:Tp, Name, [], SIWs, Threshold, Inp, RInp, Children),
    Node = node(Name, [], Inp, Children).

% if the minimal input is above the threshold, then there are no children
coopPChildren(co, _, _, _, _, T, Inp, _, []) :- Inp >= T, !.
% if the maximal input is above the threshold, then there are no children
coopPChildren(op, _, _, _, _, T, Inp, _, []) :- Inp < T, !.

% if the maximal increse of the minimal input is not sufficient, prune
coopPChildren(co, _, _, _, _, T, Inp, RInp, []) :- Inp + RInp < T, !.
% if the maximal decrease of the maximal input is not sufficient, prune
coopPChildren(op, _, _, _, _, T, Inp, RInp, []) :- Inp + RInp >= T, !.

% construct a node for the first element of the inputs
coopPChildren(CO, Tn:Tp, Prefix, PIP, [I:W|WIs], T, Inp, RInp, [Node|Sib]) :-
    Inp2 is Inp - Tn*W + Tp*W,
    RInp2 is RInp - (Tp-Tn)*W,
    coopPChildren(CO, Tn:Tp, I:Prefix, [I|PIP], WIs, T, Inp2, RInp2, Children),
    Node = node(I:Prefix, [I|PIP], Inp2, Children),
    coopPChildren(CO, Tn:Tp, Prefix, PIP, WIs, T, Inp, RInp2, Sib).
```

Figure 6.5: *Implementation to construct the pruned search trees for coalitions and oppositions: First, we check whether the perceptron is of the correct form, i.e., positive for coalitions and negative for oppositions. Then the inputs are sorted, decreasing for coalitions and increasing for oppositions. The minimal (maximal) input is computed by multiplying the sum of all weights by t^- and the remaining increase (decrease) is computed by multiplying it by $(t^+ - t^-)$. Finally the root node is expanded which recursively constructs all sub-nodes. This implementation is almost the same as for the pruned coalition search trees shown in Figure 6.3. It differs in the pre-processing and the pruning only.*

Example 6.4.7 The (negative) perceptron $\mathcal{P}_h$ from Example 6.3.1 and the resulting search trees. The full tree is shown in grey and the pruned tree in black on top of it. Oppositions are shown with grey background and oppositions in white. The annotations next to the nodes show the corresponding input pattern and the maximal input.



The corresponding set of minimal oppositions is $\mathcal{O}_h = \{\{d, f\}\}$.

2. *Let $n = \langle I, C\rangle$ be a tree node and let l be its left sibling with $\langle \mathrm{id}(l), \mathrm{var}(l), h_l, l_l\rangle$ be the BDD node constructed for l. Let $d_l = \theta - \mathrm{mci}(l)$ and let $d = d_l + (t^- - t^+) \cdot w(l) + (t^+ - t^-) \cdot w(n)$. If $d > 0$ add $\langle \mathrm{id}(n), \mathrm{var}(n), l_l, l_n\rangle$ to N with $l_n = 0$ if n has no right sibling or $l_n = \mathrm{id}(\mathrm{rs}(n))$.*
3. *For each tree node $n = \langle I, C\rangle$ in $\mathsf{PT}_\mathcal{P}$ for which none of the other cases applies, add a node $\langle \mathrm{id}(n), \mathrm{var}(n), \mathrm{id}(c), l\rangle$ to N with c being the left-most child of n and $l = 0$ if n has no right sibling or $l = \mathrm{id}(\mathrm{rs}(n))$.*

The opposition ROBDD for $\mathcal{P}$ with leaf nodes 0 and 1 and root node R *is the reduced ordered binary decision diagram* $\mathrm{opBDD}_\mathcal{P}$ *with* $\mathrm{opBDD}_\mathcal{P} := \mathrm{robdd}(\prec, 0, 1, R, N)$.

Because the algorithm to construct the coBDD and the opBDD for a given perceptron are basically the same, we present an implementation covering both. In fact, both algorithms differ in the base cases and the pre-processing only. Figure 6.6 shows a possible Prolog implementation.

6.5 Composition of Intermediate Results

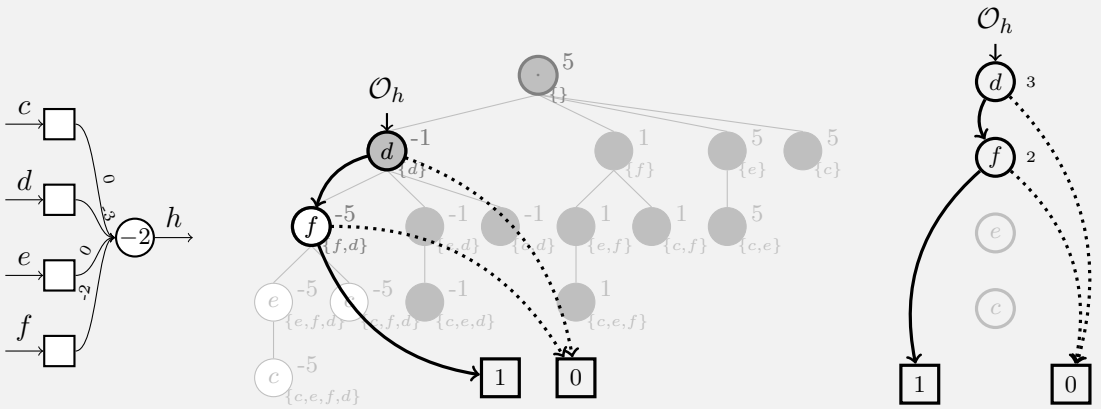
Now we show how to compose coalitions and oppositions to derive a description of the global dependencies between input and output units in the network. In the constructions above, we assumed the perceptrons to be either positive or negative. Unfortunately, this is not necessarily the case. Therefore we introduce the *positive* and *negative form* of a given perceptron. The positive form $\mathcal{P}_A^+$ of a given perceptron $\mathcal{P}_A$ is obtained by multiplying all negative weights with -1 and negating the corresponding input symbols. The negative form $\mathcal{P}_A^-$ is obtained by multiplying all positive weights with -1 and negating the corresponding inputs. Negated inputs are marked using a bar on top of it.

If we restrict t^- and t^+ to -1 and $+1$ respectively, we can conclude that a negated unit is active for all corresponding oppositions. Furthermore, we can understand negated inputs occurring in an input pattern to fix the corresponding unit to be inactive. First, we define

```
1  coopBDD(CO, Name=per(Tn:Tp, Th, IWs), B0:B1, R:BDD2) :-
2      % the perceptron must be correct, i.e., positiv for co, negativ for op
3      forall( (_:W) in IWs, correctWeight(CO, W) ),
4      % sort the inputs according to the weights
5      SIWs := sort(IWs, weightCompare(CO)),
6      VarOrder := {{ I | (I:_) in SIWs }},
7      % compute the minimal/maximal input wrt.  the empty input pattern
8      Inp := Tn*sum({{ W | (_:W) in SIWs }}), !,
9      RInp := (Tp-Tn)*sum({{ W | (_:W) in SIWs }}), !,
10     bddEmpty(VarOrder, B0, B1, BDD1),
11     coopBDD(CO:Tn:Tp:Th, Name, SIWs, Inp, RInp, nolb, BDD1, R:BDD2, _).
12
13 % If the input exceeds the Threshold, this is B1 for coalitions
14 coopBDD(co:_:_:Th, _, _, Inp, _, _, BDD, B1:BDD, D) :-
15     Inp >= Th, !, D is Inp - Th, bddNode1(BDD, B1).
16
17 % If the input is below the Threshold, this is B1 for oppositions
18 coopBDD(op:_:_:Th, _, _, Inp, _, _, BDD, B1:BDD, D) :-
19     Inp < Th, !, D is Th - Inp, bddNode1(BDD, B1).
20
21 % if the maximal increse of the minimal input is not sufficient, prune
22 coopBDD(co:_:_:Th, _, _, Inp, RInp, _, BDD, B0:BDD, nc) :-
23     Inp + RInp < Th, !, bddNode0(BDD, B0).
24
25 % if the maximal decrease of the maximal input is not sufficient, prune
26 coopBDD(op:_:_:Th, _, _, Inp, RInp, _, BDD, B0:BDD, nc) :-
27     Inp + RInp >= Th, !, bddNode0(BDD, B0).
28
29 % expand node wrt to some left neighbour if possible
30 coopBDD(O, Pref, [I:W|IWs], Inp, RInp, LBR/LBD/LBW, BDD1, R:BDD2, D) :-
31     O = _:Tn:Tp:_,
32     HD is LBD - Tp*LBW + Tn*LBW - Tn*W + Tp*W,
33     HD >= 0,
34     RInp2 is RInp - (Tp-Tn)*W,
35     bddNode(node(_, _, HR, LBR), BDD1), !,
36     coopBDD(O, Pref, IWs, Inp, RInp2, HR/HD/W, BDD1, LR:LBDD, LD),
37     composeResults(LBDD, I:Pref, HR, HD, LR, LD, R:BDD2, D).
38
39 % expand node, first go down and then go right
40 coopBDD(O, Pref, [I:W|IWs], Inp, RInp, _, BDDin, R:BDD, D) :-
41     O = _:Tn:Tp:_,
42     Inp2 is Inp - Tn*W + Tp*W,
43     RInp2 is RInp - (Tp-Tn)*W,
44     coopBDD(O, I:Pref, IWs, Inp2, RInp2, nolb, BDDin, HR:HBDD, HD),
45     (   HD = nc % i.e., we can cut all right siblings
46     ->  ( bddNode0(BDD, R), BDD=BDDin, D=nc )
47     ;   ( coopBDD(O, Pref, IWs, Inp, RInp2, HR/HD/W, HBDD, LR:LBDD, LD),
48           composeResults(LBDD, I:Pref, HR, HD, LR, LD, R:BDD, D) )).
```

Figure 6.6: *Implementation to construct the reduced ordered BDD for coalition and the reduced ordered BDD for oppositions for a given perceptron $\mathcal{P}$: After checking whether the perceptron has the correct form, i.e., the positive for the coalitions and negative for the oppositions, the inputs are ordered wrt. their weights, i.e., in both cases descending wrt. the absolute values. The first and second clauses for* `coopBDD/9` *capture the two different base cases if a coalition or an opposition is found. The third and fourth clause prune the search if there is no solution any more and the remaining two clauses are exactly as in Figure 6.4.*

Example 6.4.8 The perceptron $\mathcal{P}_h$ from Example 6.3.1 is shown on the left. Again, we show the search tree from above and overlayed the resulting BDD in the middle and the final pure BDD on the right.



The corresponding set of minimal oppositions is $\mathcal{O}_h = \{\{d, f\}\}$ and the corresponding path from root to 1 is $[3, 2, 1]$. The pure BDD on the right is annotated with the node ids used in the paths. For variables not occurring in the BDD, we added grey nodes in the BDD to show the full variable order.

the positive and negative form of a perceptron formally and then show how to compose the intermediate results.

6.5.1 The Positive and Negative Form of a Perceptron

As mentioned above, the positive form is obtained by multiplying all negative weights by -1 and inverting the corresponding input symbol. Figure 6.7 shows the Prolog implementation and Example 6.5.1 the positive forms of the perceptrons from Example 6.3.1.

Definition 6.5.1 (Positive Form) *Let $\mathcal{P}_A = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a perceptron. Its positive form $\mathcal{P}_A^+$ is defined as follows:*

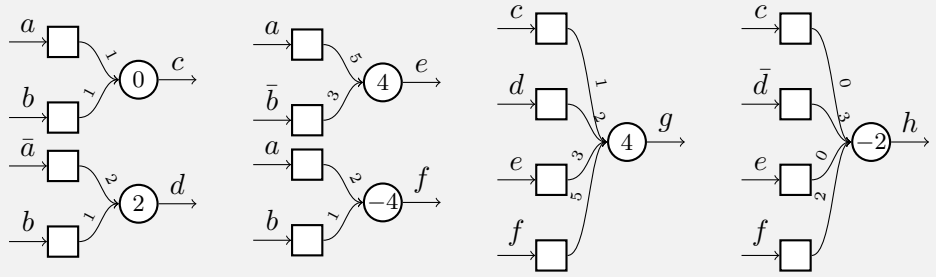
$$\mathcal{P}_A^+ := \langle t^-, t^+, \theta, \mathcal{I}', \omega' \rangle \quad \textit{with}$$
$$\mathcal{I}' = \{\bar{U} \mid U \in \mathcal{I} \textit{ and } \omega(U) < 0\} \cup \{U \mid U \in \mathcal{I} \textit{ and } \omega(U) \geq 0\} \quad \textit{and}$$
$$\omega' : \mathcal{I}' \to \mathbb{R} : U \mapsto \begin{cases} \omega(U) & \textit{if } U \in \mathcal{I} \\ -\omega(U) & \textit{otherwise (i.e., } U = \bar{A} \textit{ for some } A \in \mathcal{I}) \end{cases}$$

```
1 posPerceptron(A=per(TNP,T,IW), A=per(TNP,T,PIW)) :-
2         PIW := { I:W | (I:W) in IW, W >= 0 } union
3                { -I:PW | (I:W) in IW, W < 0, PW := -W }.
```

Figure 6.7: *Implementation to construct the positive form of a given perceptron.*

The negative form of a perceptron is obtained analogously, by multiplying all positive weights by -1 and inverting the corresponding input symbols. The Prolog implementation is shown in Figure 6.8 and Example 6.5.2 shows all negative perceptrons for our running example.

Example 6.5.1 The positive forms $\mathcal{P}_c^+, \ldots, \mathcal{P}_h^+$ of the perceptrons from Example 6.3.1:



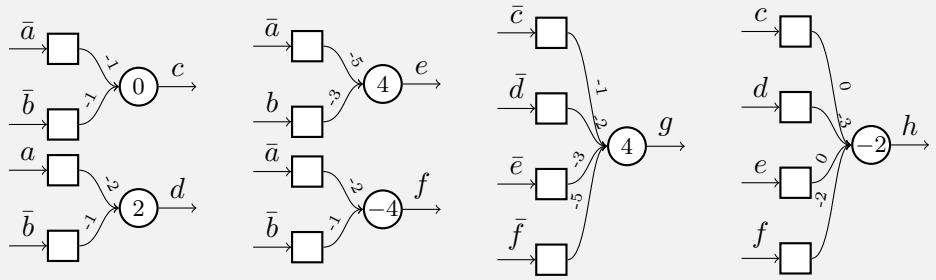
Definition 6.5.2 (Negative Form) *Let $\mathcal{P}_A = \langle t^-, t^+, \theta, \mathcal{I}, \omega \rangle$ be a perceptron. Its negative form $\mathcal{P}_A^-$ is defined as follows:*

$$\mathcal{P}_A^- := \langle t^-, t^+, \theta, \mathcal{I}', \omega' \rangle \quad \textit{with}$$
$$\mathcal{I}' = \{U \mid U \in \mathcal{I} \textit{ and } \omega(U) \leq 0\} \cup \{\bar{U} \mid U \in \mathcal{I} \textit{ and } \omega(U) > 0\} \quad \textit{and}$$
$$\omega' : \mathcal{I}' \to \mathbb{R} : U \mapsto \begin{cases} \omega(U) & \textit{if } U \in \mathcal{I} \\ -\omega(U) & \textit{otherwise (i.e., } U = \bar{A} \textit{ for some } A \in \mathcal{I}) \end{cases}$$

```
1 negPerceptron(A=per(TNP,T,IW), A=per(TNP,T,NIW)) :-
2         NIW := { I:W | (I:W) in IW, W =< 0 } union
3                { -I:PW | (I:W) in IW, W > 0, PW := -W }.
```

Figure 6.8: *Implementation to construct the negative form of a given perceptron.*

Example 6.5.2 The negative forms $\mathcal{P}_c^-, \ldots, \mathcal{P}_h^-$ of the perceptrons from Example 6.3.1:

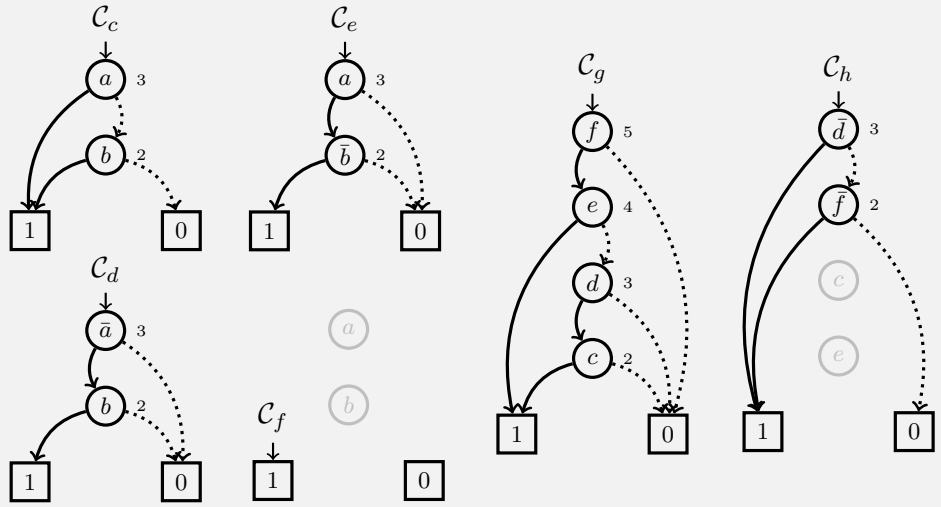


We can now use the algorithm shown in Figure 6.6 to compute the corresponding coalition and opposition ROBDDs. Those are shown in Example 6.5.3 and 6.5.4, respectively. The resulting sets of minimal coalitions and oppositions are given in Example 6.5.5.

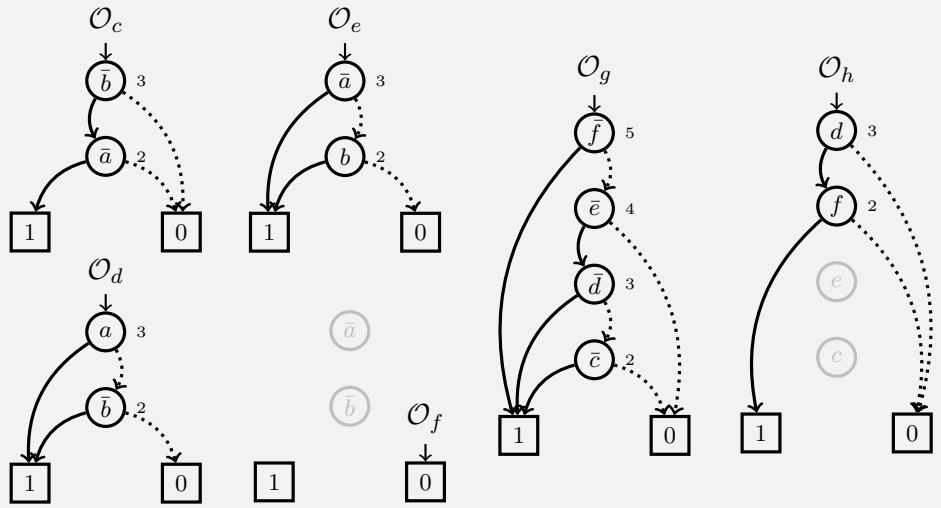
6.5.2 Naive Composition of Coalitions and Oppositions

As mentioned above, the sets of minimal coalitions and oppositions can be used to describe the behaviour of a perceptron in a declarative way, by giving necessary and sufficient condi-

Example 6.5.3 The coalition BDD for all positive perceptrons shown in Example 6.5.1:



Example 6.5.4 The opposition BDD for all negative perceptrons shown in Example 6.5.2:



Example 6.5.5 Minimal coalitions and oppositions of all units from Figure 6.2.1.

Unit	Minimal Coalitions	Minimal Oppositions
c	$\{\{a\},\{b\}\}$	$\{\{\bar{a},\bar{b}\}\}$
d	$\{\{\bar{a},b\}\}$	$\{\{a\},\{\bar{b}\}\}$
e	$\{\{a,\bar{b}\}\}$	$\{\{\bar{a}\},\{b\}\}$
f	$\{\{\}\}$	$\{\}$
g	$\{\{e,f\},\{c,d,f\}\}$	$\{\{\bar{f}\},\{\bar{d},\bar{e}\},\{\bar{c},\bar{e}\}\}$
h	$\{\{\bar{d}\},\{\bar{f}\}\}$	$\{\{d,f\}\}$

tions that turn a perceptron active and inactive. Furthermore, both sets can be turned into propositional formulae representing the same idea.

Definition 6.5.3 *Let A be a perceptron and let $\mathcal{C}_A = \{\{c_{1_1}, \ldots, c_{n_1}\}, \ldots, \{c_{m_1}, \ldots, c_{n_m}\}\}$ be the set of minimal coalitions. We define the corresponding propositional formula* $\mathrm{pf}(\mathcal{C}_A)$ *as follows:*

$$\mathrm{pf}(\mathcal{C}_A) = ((c_{1_1} \wedge \ldots \wedge c_{n_1}) \vee \ldots \vee (c_{m_1} \wedge \ldots \wedge c_{n_m}))$$

Analogously, we define the propositional formula corresponding to a set of minimal oppositions $\mathcal{O}_A = \{\{o_{1_1}, \ldots, o_{n_1}\}, \ldots, \{o_{m_1}, \ldots, o_{n_m}\}\}$ as:

$$\mathrm{pf}(\mathcal{O}_A) = ((o_{1_1} \wedge \ldots \wedge o_{n_1}) \vee \ldots \vee (o_{m_1} \wedge \ldots \wedge o_{n_m}))$$

After computing all sets of minimal coalitions and oppositions as in Example 6.5.5 above, we could use the corresponding propositional formulae to compose those sets and obtain the global dependencies between input and output units. This can be done, by taking the formula corresponding to the coalitions of some output unit and recursively replacing all propositional variables with their corresponding formula as introduced in Definition 6.5.3. This naive composition is shown in Example 6.5.6. This is not necessarily the best approach, because some of those sets might be irrelevant and, hence, we do not need to compute them. In the following section we show how to compose the intermediate results taking advantage of their representation as BDDs.

Example 6.5.6 From $\mathcal{C}_g = \{\{e, f\}, \{c, d, f\}\}$, we can derive the propositional formula $g \leftarrow (e \wedge f) \vee (c \wedge d \wedge f)$. Analogously, we obtain:

$$\begin{array}{ll}
c \leftrightarrow (a \vee b) & \bar{c} \leftrightarrow (\bar{a} \wedge \bar{b}) \\
d \leftrightarrow (\bar{a} \wedge b) & \bar{d} \leftrightarrow (a \vee \bar{b}) \\
e \leftrightarrow (a \wedge \bar{b}) & \bar{e} \leftrightarrow (\bar{a} \vee b) \\
f \leftrightarrow \mathit{true} & \bar{f} \leftrightarrow \mathit{false} \\
g \leftrightarrow ((e \wedge f) \vee (c \wedge d \wedge f)) & \bar{g} \leftrightarrow (\bar{f} \vee (\bar{d} \wedge \bar{e}) \vee (\bar{c} \wedge \bar{e})) \\
h \leftrightarrow (\bar{d} \vee \bar{f}) & \bar{h} \leftrightarrow (d \wedge f)
\end{array}$$

Using those dependecies, we obtain

$$\begin{array}{ll}
g \leftrightarrow ((e \wedge f) \vee (c \wedge d \wedge f)) & h \leftrightarrow (\bar{d} \vee \bar{f}) \\
\quad \leftrightarrow (((a \wedge \bar{b}) \wedge \mathit{true}) \vee ((a \vee b) \wedge (\bar{a} \wedge b) \wedge \mathit{true})) & \quad \leftrightarrow ((a \vee \bar{b}) \vee \mathit{false}) \\
\quad \leftrightarrow ((a \wedge \bar{b}) \vee ((a \vee b) \wedge \bar{a} \wedge b)) & \quad \leftrightarrow (a \vee \bar{b}) \\
\quad \leftrightarrow ((a \wedge \bar{b}) \vee (\bar{a} \wedge b)) &
\end{array}$$

6.5.3 Composition of Coalition- and Opposition-BDDs

Now we show how to obtain a reduced ordered BDD representing the global dependencies between input and output nodes of a given network. This is done by composing the intermediate results represented as BDDs. In principle there are two approaches. On the one hand, we could simply replace a node with the corresponding BDD. Unfortunately, this leads to non-ordered and non-reduced BDDs as shown in Example 6.5.7. The resulting BDD could afterwards be reduced and ordered. On the other hand, we can maintain a reduced ordered BDD while

replacing non-input nodes by their corresponding BDDs. We pursue this second approach in the sequel.

During the extraction, we incrementally build a reduced ordered BDD, which we referred to as *global BDD*. The algorithm is based on the *expansion* of nodes into this global BDD. During this expansion, we replace nodes within a BDD, that do not correspond to input units of the network, by their coalitions or oppositions BDDs. This is done until only variables occur that correspond to input units of the network. We first define the expansion formally and then present the algorithm.

Definition 6.5.4 *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network and let $A \in \mathcal{U} \setminus \mathcal{U}_{\mathsf{inp}}$ be a non-input node within $\mathcal{N}$. Let $B = \langle \prec, 0, 1, R, N \rangle$ be a given ROBDD. The* coalition expansion of A into B wrt. B_0 and $B_1 \in N$ *is the coalition BDD $\langle \prec_A, B_0, B_1, R_A, N_A \rangle$ with leaf nodes B_0 and B_1 as introduced in Definition 6.4.9. The* opposition expansion of A into B wrt. B_0 and $B_1 \in N$ *is the opposition BDD $\langle \prec_A, B_0, B_1, R_A, N_A \rangle$ with leaf nodes B_0 and B_1 as introduced in Definition 6.4.13. The* full expansion of A *is the reduced ordered BDD B_A obtained from the coalition BDD* coBDD_A *by expanding all nodes $\langle i, B, h, l \rangle$ for $B \in \mathcal{U} \setminus \mathcal{U}_{\mathsf{inp}}$ using the coalition expansion of B wrt. h and l, all nodes $\langle i, \bar{B}, h, l \rangle$ using the opposition expansion of B wrt. h and l and by replacing all nodes $\langle i, \bar{C}, h, l \rangle$ by $\langle i, C, l, h \rangle$ for $C \in \mathcal{U}_{\mathsf{inp}}$.*

Theorem 6.5.5 *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network and let B_A be the full expansion of some $A \in \mathcal{U} \setminus \mathcal{U}_{\mathsf{inp}}$. Then we find B_A to be a propositional representation of A as introduced in Definition 6.1.3.*

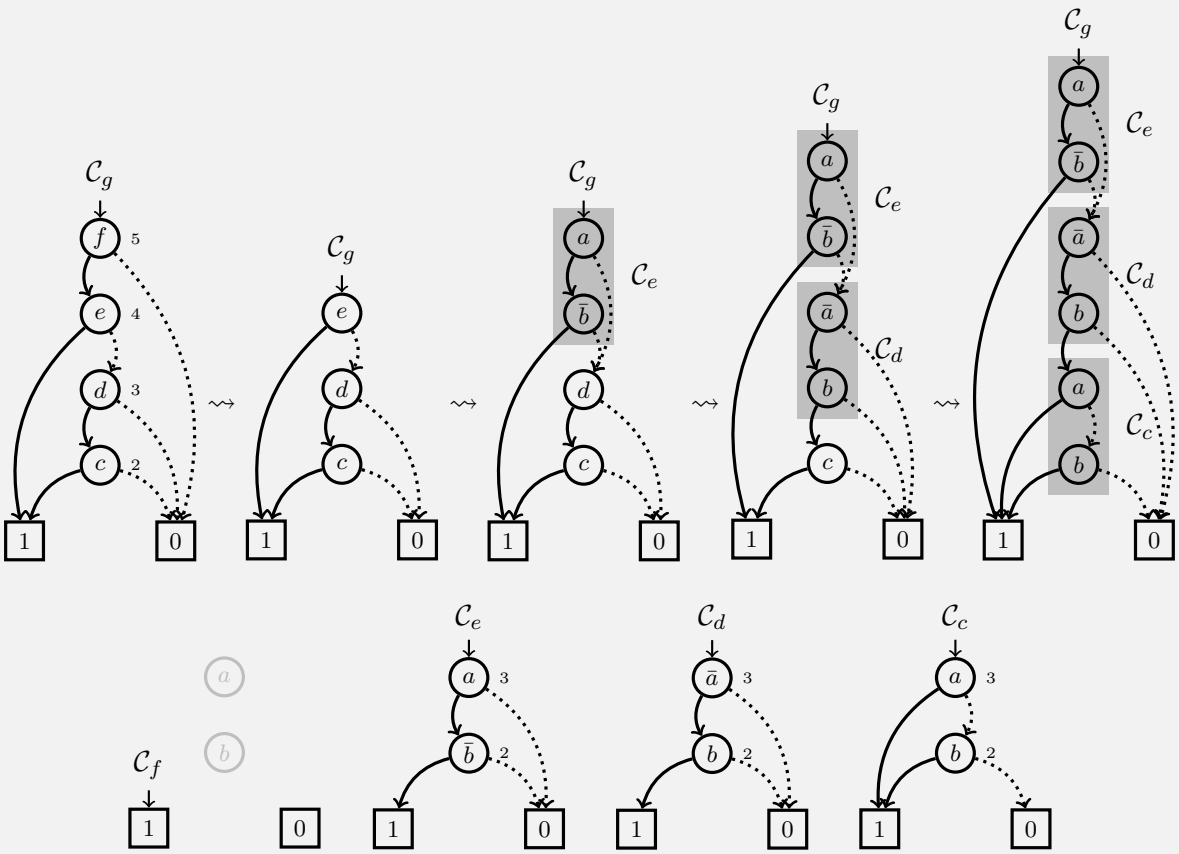
> **Proof** All variables occurring in the final BDD correspond to input nodes, i.e., the resulting BDD is by construction a propositional formula over $\mathcal{U}_{\mathsf{inp}}$. It remains to be shown that all models of B_A activate the unit A, if propagated through the network. This can be shown by induction on the maximal distance of A to the input units. If this distance is 1, e.g., all predecessors of A are input units, we find B_A to be the coalition BDD for A and, hence, that all models activate the unit (see Lemma 6.4.11). If the distance is larger, we find all units occurring positively in the coalition BDD for A to be replaced by the corresponding coalition BDD and all occurring negated to be replaced by their opposition BDDs, which correspond to equivalent logic formulae. □

Figure 6.9 shows an algorithm to expand a node into a given global BDD. Starting from the given BDD, the node is expanded by computing its coalition or opposition BDD. All nodes not corresponding to input units are expanded further. The algorithm consists of three parts:

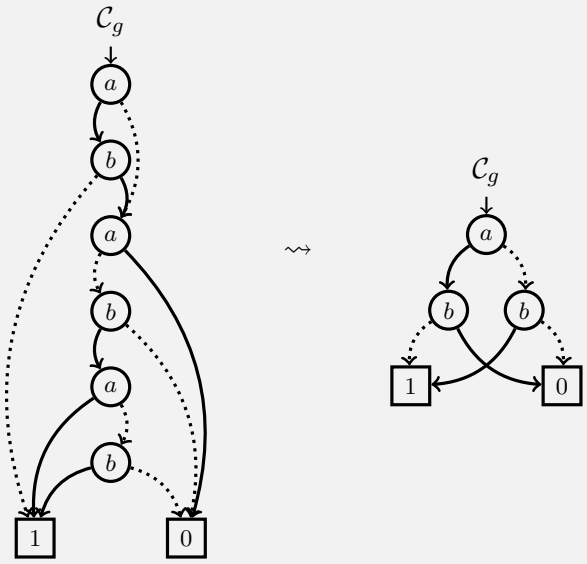
1. *Expansion of a given node* (`coopExpand/9`): After constructing the correct form of the corresponding perceptron, the BDD representing its oppositions or coalitions is extracted using `coopBDD/4` as shown in Figure 6.6. The first clause of `coopExpand/9` captures the case that the high and low branch coincide. In this case the result is equal to these nodes and no expansion need to be done. Afterwards, the resulting BDD is incorporated into the global BDD using `coopInc/7` as described below.

2. *Incorporation of a BDD into the global BDD* (`coopInc/7`): The incorporation of some node `R` of the given `BDD` into `BDDa` is done by first incorporating high and low branch of `R` and then calling `coopIncNode/8` for the results. The first and second clause of `coopInc/7` capture the cases that the node `R` is the high or low node of the corresponding BDD. In this case the result is simply `R`, because it already belongs to `BDDa`.

Example 6.5.7 A simple composition by replacing nodes with their corresponding BDDs. The first row shows the intermediate results while replacing the nodes. The second row shows the BDDs which have been inserted. E.g., the node for e has been replaced by coBDD$_e$.



Because $\bar{b}$ denotes the negation of b, we can simply swap high and low branch of all nodes with negated variables and obtain the BDD shown below on the left. After reducing and ordering this BDD, we obtain the final result as shown on the right.



3. *Incorporation of a node into the global BDD* (`coopIncNode/8`): The final BDD must contain only nodes for variables corresponding to input nodes of the network. Those are listed in `IIDs`. While adding a node for variable `V` branching to `High` and `Low`, we need to distinguish 4 cases:
   - *The variable corresponds to an input of the network*: The node is constructed using the if-then-else operator `ite` within the BDD.
   - *The variable corresponds to the negation of some input of the network*: As before, but high and low branch are exchanged.
   - *The variable occurs negated*: The node is constructed by expanding the corresponding opposition BDD using `coopExpand/9`.
   - *The variable occurs positive*: The node is constructed by expanding the corresponding coalition BDD.

Using this expansion algorithm, we can formulate the full extraction of a feed-forward threshold network as presented in Figure 6.10. After decomposing the given network into perceptrons, the set of input `IIDs` and output units `OIDs` are computed. A variable order is fixed and the global BDD initialised to be empty. Finally, all output nodes are expanded into the global BDD by calling `coopExtraction/8`. Because all output units are connected to the same input and hidden units, we expand them into the same BDD. This global BDD has therefore multiple root nodes, namely one for each of the output units. `BDDb` is unified with the resulting global BDD and `ORs` with a list of ("output unit"="root node")-pairs.

Example 6.5.8 shows the call to `coopExtraction/5` for the network from Figure 6.2.1. It also shows the resulting BDD. Please note, that some nodes contained in the BDD are not really required (and not shown in the graphical representation), but are results of the construction. E.g., the node `node(a,0,3,11)`. More advanced implementations remove those intermediate nodes already during the construction.

In the following section, we show how to incorporate integrity constraints. As for the composition, the representation as BDDs gives raise to a very natural treatment of the problem.

6.6 Incorporation of Integrity Constraints

As mentioned above, integrity constraints restrict the set of possible inputs of a given network. Usually only a subset of all possible input combinations is interesting for a user of a neural network, and even more important the training samples are usually taken from this subset only. Therefore, the network learns to solve the task under the implicit conditions imposed by the selection of the training pattern. Integrity constraints are a method to make those conditions explicit. In the sequel, we show how to incorporate integrity constraints into our approach. But first, we discuss a small example to further motivate their usage.

The task in the encoder-decoder problem, as given in Example 6.6.1, is to learn a one-to-one mapping from inputs to outputs. In all training patterns, exactly one input unit is active, while all others are inactive. Please note, this fact is an integrity constrained and can be incorporated into the extraction. While using less hidden than input units, the network is forced to learn a compressed representation within the hidden layer. The network shown in Example 6.6.2 solves the task correctly, i.e., all inputs are mapped to the correct outputs. But applying the extraction procedure described above, yields an unwanted results. The resulting

```
% coopExpand(Perceptrons, IIDs, BDDa, COOP, O, BDD0, BDD1, R, BDDb)
% Unifies R with the COOP-expansion of O wrt.  BDD0 and BDD1 into BDDa.

% If BDD0 and BDD1 coincide, the result will be that node
coopExpand(_Perceptrons, _IIDs, BDDa, _CO, _O, BDD01, BDD01, BDD01, BDDa) :- !.
% Expand O as a coalition.  First the positive form of the perceptron is
% constructed, then the coalition BDD construncted, such that its leaf nodes
% are BDD0 and BDD1.  Finally, the result is incorporated into the BDDa and
% the result returned as R and BDDb.
coopExpand(Perceptrons, IIDs, BDDa, co, O, BDD0, BDD1, R, BDDb) :-
        member(O=P, Perceptrons),
        posPerceptron(O=P, O=Perceptron),
        coopBDD(co, O=Perceptron, BDD0:BDD1, RootNode:OBDD),
        coopInc(Perceptrons, IIDs, BDDa, OBDD, RootNode, R, BDDb).
% ...  as above for oppositions
coopExpand(Perceptrons, IIDs, BDDa, op, O, BDD0, BDD1, R, BDDb) :-
        member(O=P, Perceptrons),
        negPerceptron(O=P, O=Perceptron),
        coopBDD(op, O=Perceptron, BDD0:BDD1, RootNode:OBDD),
        coopInc(Perceptrons, IIDs, BDDa, OBDD, RootNode, R, BDDb).

% coopInc(Perceptrons, IIDs, BDDa, BDD, R, R2, BDDb)
% Incorporation of the node R in BDD into the bdd BDDa.  R2 and BDDb are
% unified with the result.

% If R is one of the leaf node, this node will be the result, because it
% already is a node within BDDa.
coopInc(_Perceptrons, _IIDs, BDDa, BDD, R, R, BDDa) :- bddNode0(BDD, R), !.
coopInc(_Perceptrons, _IIDs, BDDa, BDD, R, R, BDDa) :- bddNode1(BDD, R), !.
% ...  if not, incorporate the high and low branch of R and call coopIncNode
coopInc(Perceptrons, IIDs, BDDa, BDD, R, R2, BDDd) :-
        bddNode(BDD, node(V, H, L, R)), !,
        coopInc(Perceptrons, IIDs, BDDa, BDD, H, H2, BDDb),
        coopInc(Perceptrons, IIDs, BDDb, BDD, L, L2, BDDc),
        coopIncNode(Perceptrons, IIDs, BDDc, V, H2, L2, R2, BDDd).

% coopIncNode(Perceptrons, IIDs, BDDa, V, High, Low, R, BDDb)
% R and BDDb are unified with the result of incorporating of some
% node (V -> High ; Low) into the BDDa.

% If V is a member if IIDs, just apply the if-then-else operator within BDDa
coopIncNode(_Perceptrons, IIDs, BDDa, V, High, Low, R, BDDb) :-
        member(V, IIDs), !,
        bddApply(BDDa, ite, V, High, Low, R, BDDb).
% If V is a negation of some element of IIDs, apply the ite reversed
coopIncNode(_Perceptrons, IIDs, BDDa, -V, High, Low, R, BDDb) :-
        member(V, IIDs), !,
        bddApply(BDDa, ite, V, Low, High, R, BDDb).
% If V is negated, expand it as opposition into BDDa
coopIncNode(Perceptrons, IIDs, BDDa, -V, High, Low, R, BDDb) :- !,
        coopExpand(Perceptrons, IIDs, BDDa, op, V, Low, High, R, BDDb).
% If V is not negated, expand it as coalition into BDDa
coopIncNode(Perceptrons, IIDs, BDDa, V, High, Low, R, BDDb) :-
        coopExpand(Perceptrons, IIDs, BDDa, co, V, Low, High, R, BDDb).
```

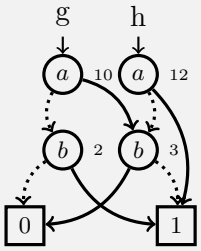
Figure 6.9: *The expansion of some node* `O` *into* `BDDa`*: The expansion is done wrt. a set of inputs* `IIDs` *and a high* `BDD1` *and low node* `BDD0` *within* `BDDa`*.*

```
coopExtraction(N, IIDs, OIDs, ORs, BDDb) :-
        % decompose the network into perceptrons
        tffn2perceptrons(N, Perceptrons),
        % find all input and output unit IDs
        N = tffn(_U, Ui, Uo, _C, -1:1),
        IIDs := { ID | unit(ID,_) in Ui},
        OIDs := { ID | unit(ID,_) in Uo},
        % fix a variable order
        VO = IIDs,
        % fix initialise the BDD
        bddEmpty(VO, 0, 1, BDDa),
        % start the extraction of the output nodes
        coopExtraction(Perceptrons, BDDa, 0, 1, IIDs, OIDs, ORs, BDDb).

coopExtraction(_Perceptrons, BDDa, _BDD0, _BDD1, _IIDs, [], [], BDDa).
coopExtraction(Perceptrons, BDDa, BDD0, BDD1, IIDs, [O|Os], [O=R|ORs], BDDc) :-
        coopExpand(Perceptrons, IIDs, BDDa, co, O, BDD0, BDD1, R, BDDb),
        coopExtraction(Perceptrons, BDDb, BDD0, BDD1, IIDs, Os, ORs, BDDc).
```

Figure 6.10: *The extraction of a reduced ordered* `BDDb` *representing the coalitions of all output nodes wrt. the input nodes for a given network* `N`*: Please note that the final BDD has multiple root nodes, one for each output unit.*

Example 6.5.8 The call to `coopExtraction/5` as presented in Figure 6.10. The resulting BDDs is shown on the right. Please note, that only those nodes are shown, which are relevant for the root nodes corresponding to g and h. E.g., node number 11, is not shown, even though it is part of the result of our naive BDD implementation.

```
?- Net = tffn([unit(a, 0), unit(b, 0), ...],
              [unit(a, 0), unit(b, 0)],
              [unit(g, 4), unit(h, -2)],
              [a/c/1, a/d/ -2, ...],
              -1:1),
   coopExtraction(Net, IIDs, OIDs, ORs, BDD).

IIDs = [a, b]
OIDs = [g, h]
ORs = [g=10, h=12]
BDD = bdd([0, 1, a, b], 0, 1,
      [node(a, 1, 3, 12), node(a, 0, 3, 11),
       node(a, 3, 2, 10), node(a, 3, 0, 9),
       node(a, 3, 1, 8), node(a, 1, 2, 7),
       node(a, 0, 2, 6), node(a, 0, 1, 5),
       node(a, 1, 0, 4), node(b, 0, 1, 3),
       node(b, 1, 0, 2)]),
```

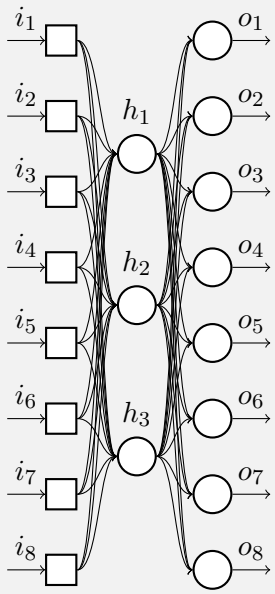


Example 6.6.1 The encoder-decoder problem is to learn a 1-to-1 mapping between the input and output layer under the condition that exactly one unit is active at a time. The table below shows the full set of input-output pairs.

input pattern	output pattern
$(+1,-1,-1,-1,-1,-1,-1,-1)$	$(+1,-1,-1,-1,-1,-1,-1,-1)$
$(-1,+1,-1,-1,-1,-1,-1,-1)$	$(-1,+1,-1,-1,-1,-1,-1,-1)$
$(-1,-1,+1,-1,-1,-1,-1,-1)$	$(-1,-1,+1,-1,-1,-1,-1,-1)$
$(-1,-1,-1,+1,-1,-1,-1,-1)$	$(-1,-1,-1,+1,-1,-1,-1,-1)$
$(-1,-1,-1,-1,+1,-1,-1,-1)$	$(-1,-1,-1,-1,+1,-1,-1,-1)$
$(-1,-1,-1,-1,-1,+1,-1,-1)$	$(-1,-1,-1,-1,-1,+1,-1,-1)$
$(-1,-1,-1,-1,-1,-1,+1,-1)$	$(-1,-1,-1,-1,-1,-1,+1,-1)$
$(-1,-1,-1,-1,-1,-1,-1,+1)$	$(-1,-1,-1,-1,-1,-1,-1,+1)$

final BDD for the output unit o_1 is shown in Example 6.6.3 on the left. A glimpse on the final result after incorporation of the corresponding integrity constraint is also shown on the right.

Example 6.6.2 An encoder-decoder network which solves the task correctly. The structure is shown on the left and the weight matrices including the thresholds are shown on the right.



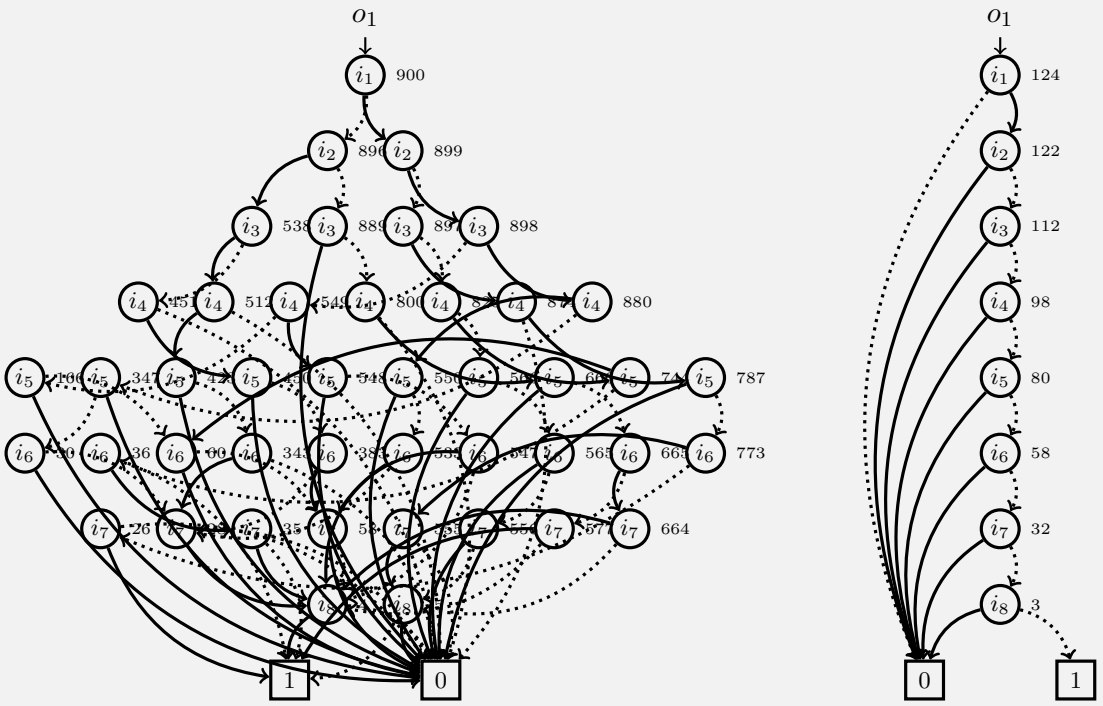
	$\mathbf{h_1}$	$\mathbf{h_2}$	h_3
$\mathbf{i_1}$	−2.37	−0.39	2.72
$\mathbf{i_2}$	4.52	3.77	4.56
$\mathbf{i_3}$	4.11	−7.03	−7.19
$\mathbf{i_4}$	1.38	−6.06	6.23
$\mathbf{i_5}$	3.21	6.21	−3.95
$\mathbf{i_6}$	−3.84	3.69	−3.25
$\mathbf{i_7}$	−4.33	4.37	3.18
$\mathbf{i_8}$	−4.64	−2.71	−3.59
θ	−0.46	0.07	−0.08

	$\mathbf{o_1}$	$\mathbf{o_2}$	$\mathbf{o_3}$	$\mathbf{o_4}$	$\mathbf{o_5}$	$\mathbf{o_6}$	$\mathbf{o_7}$	o_8
$\mathbf{h_1}$	−1.49	1.91	2.05	1.47	1.72	−1.59	−1.90	−1.76
$\mathbf{h_2}$	−1.49	1.58	−1.99	−1.87	2.19	1.78	1.77	−1.52
$\mathbf{h_3}$	1.49	2.15	−1.77	2.05	−1.58	−1.94	1.74	−1.90
θ	−2.96	−4.12	−4.30	−3.92	−3.83	−3.84	−3.92	−3.68

One approach to incorporate integrity constraints is the construction of the full BDD as done above, and then to refine it using the constraints. But, we can also apply the constraints while expanding the BDD. This can be done by extracting the network into a BDD representing the integrity constraints. If a rule is extracted which contradicts the constraint then the expansion yields the 0 of this global BDD. Therefore, only those rules are extracted that are consistent with the integrity constraints.

Definition 6.6.1 *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network and let $A \in \mathcal{U} \setminus \mathcal{U}_{\mathsf{inp}}$ be a non-input node within $\mathcal{N}$. Let* IC *be a propositional integrity constraint and let $B_{\mathrm{IC}} = \langle \prec, B_0, B_1, R_{\mathrm{IC}}, N \rangle$ be the corresponding ROBDD. The* full expansion of A wrt. IC *is the reduced ordered BDD B_A obtained from the coalition expansion of A wrt. R_{IC} and B_0 by expanding all non-input nodes as in Definition 6.5.4.*

Example 6.6.3 The full extraction of the output unit o_1 from the trained encoder-decoder network are shown below. On the left the result of the extraction without integrity constraints is shown and on the right the result after incorporating the constraint that at most one input unit is active.



The numbers right to the nodes show their internal IDs. I.e., to construct the BDD shown on the left 900 nodes have been constructed, while only 124 are necessary to construct the BDD incorporating the integrity constraint.

Figure 6.11 shows the implementation of the full extraction incorporating the integrity constraints. Applying the algorithm to the network from Example 6.6.2 yields the BDD shown in Example 6.6.4. The BDD on the left represents the corresponding integrity constraint stating that at most one input unit can be active. The final BDD on the right contains the propositional representation wrt. this constraint of all output nodes. Please note that all links pointing to 0 are omitted for clarity. Looking at the extraction of o_4, we find a "gap", because the node for input i_4 is missing. The extraction for o_4 states that the unit is active, as soon as all inputs $i_1, \ldots, i_3$ and $i_5, \ldots, i_8$ are inactive. This is not intended but sufficient to classify all inputs correctly. Please note, that changing the integrity constraint from max(1) to exactly(1) would fill this gap. The usage of the "softer" max(1)-constraint shows that the integrity constraints should be selected carefully. But even though we do not know all constraints, the extraction results are nonetheless better as the ones obtained using no constraints at all.

```
1  coopExtractionICs(N, ICs, IIDs, OIDs, ORs, BDDc) :-
2          % decompose the network into perceptrons
3          tffn2perceptrons(N, Perceptrons),
4          % find all input and output unit IDs
5          N = tffn(_U, Ui, Uo, _C, -1:1),
6          IIDs := { ID | unit(ID,_) in Ui },
7          OIDs := { ID | unit(ID,_) in Uo },
8          % fix a variable order
9          VO = IIDs,
10         % initialise the BDD
11         bddEmpty(VO, 0, 1, BDDa),
12         % create the BDD representing the integrity constraints
13         icBDD(BDDa, ICs, ICRoot, BDDb),
14         % start the extraction of the output nodes
15         coopExtraction(Perceptrons, BDDb, 0, ICRoot, IIDs, OIDs, ORs, BDDc).
```

Figure 6.11: *The full extraction into a single reduced ordered BDD including the incorporation of integrity constraints for all output nodes.*

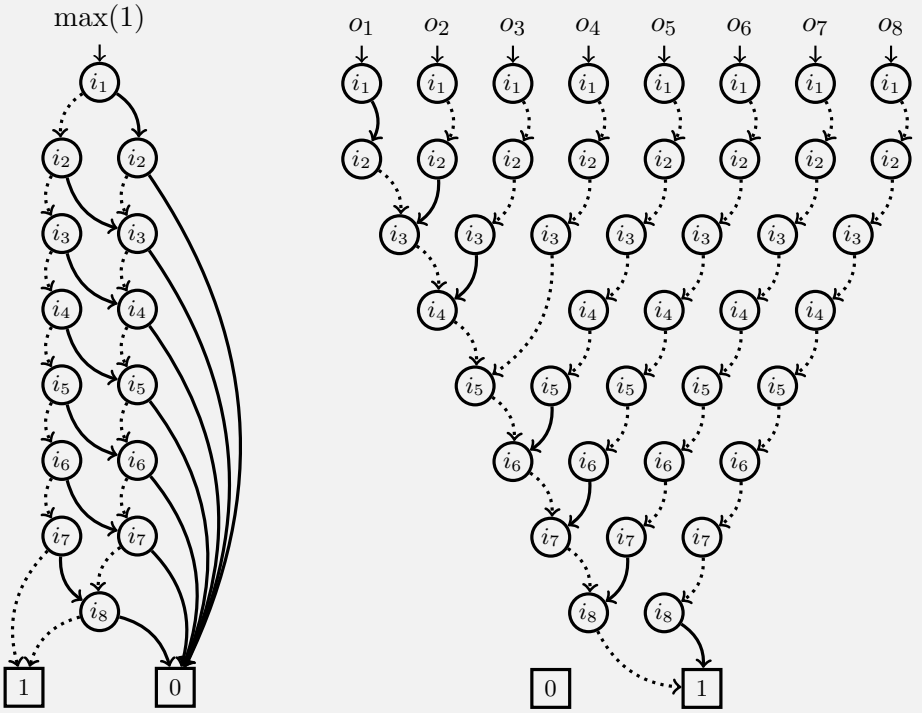
Theorem 6.6.2 *Let $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+ \rangle$ be a threshold feed-forward network, let $A \in \mathcal{U} \setminus \mathcal{U}_{\mathsf{inp}}$ be a non-input node within $\mathcal{N}$ and let* IC *be a propositional integrity constraint. Let B_A be the full expansion of A and let B'_A be the full expansion of A wrt.* IC *as introduced in Definition 6.6.1. Then we find $B'_A \equiv B_A \wedge \mathrm{IC}$ and, hence, B'_A to be a propositional representation wrt.* IC*.*

Proof Because the unit is expanded into a BDD representing the integrity constraint IC, we find $I \not\models B'_A$ for all $I \not\models \mathrm{IC}$. For all those I with $I \models \mathrm{IC}$, we find B_{IC} to be equal to $\top$ and $(B'_A)^I = (B_A)^I$. Using Lemma 6.1.7, we can conclude that B'_A is a propositional representation of A wrt. IC. □

6.7 Extraction of Propositional Logic Programs

To complete the cycle shown in Figure 1.1 on page 4, we transform the final BDD into a logic program. This is done by following all paths from the root of the BDD B_A extracted for some unit A to the terminal 1. All paths represent combinations of inputs that activate unit A. If

Example 6.6.4 The BDD on the left represents the integrity constraint for the encoder-decoder example. It states that at most one input unit can be active at a time. The BDD on the right shows the extraction of all output nodes wrt. this integrity constraint. Please note that all links pointing to 0 are left out for clarity.



the path follows a high branch then the corresponding input must be active. If it follows a low branch then the input must be inactive. This can easily be represented as a propositional clause.

Definition 6.7.1 (Corresponding Logic Program) *Let $B_A = \langle \prec, 0, 1, R, N\rangle$ be the full extraction of some output node $A \in \mathcal{U}_{\mathsf{out}}$ from $\mathcal{N} = \langle \mathcal{U}, \mathcal{U}_{\mathsf{inp}}, \mathcal{U}_{\mathsf{out}}, \mathcal{C}, \omega, \theta, t^-, t^+\rangle$. Let $p = \{p_1, \ldots, p_n\}$ be the set of all paths from the root of B_A to 1 as introduced in Definition 2.4.4. Then we define the* corresponding logic program P_A *as follows:*

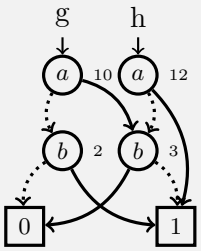
$$P_A = \{P_A(p_k) \mid p_k \in p\} \quad \textit{with}$$

$$P_A([i_1, \ldots, i_c]) = A \leftarrow B_1 \wedge \ldots \wedge B_{c-1} \quad \textit{and} \quad B_j = \begin{cases} \mathrm{var}(i_j) & \textit{if } i_{j+1} = \mathrm{high}(i_j) \\ \neg\, \mathrm{var}(i_j) & \textit{if } i_{j+1} = \mathrm{low}(i_j) \end{cases}$$

Lemma 6.7.2 *Let B_A be a BDD and let P_A be the corresponding logic program as introduced above. Then there exists a clause $A \leftarrow Body \in P_A$ with $I \models Body$ for exactly those I with $I \models B_A$.*

Proof This lemma follows by definition of P_A and the fact that the set of all paths from the root to 1 represents all models of the given BDD. □

Example 6.7.1 Considering the BDD from Example 6.5.8, we obtain the following set of paths to 1 and the corresponding logic programs as introduced in Definition 6.7.1:



$$p_g := \{[10,2,1] \quad [10,3,1]\ \} \qquad P_g := \{g \leftarrow \neg a \wedge b \quad g \leftarrow a \wedge \neg b\}$$
$$p_h := \{[12,3,1] \quad [12,1]\ \} \qquad P_h := \{h \leftarrow \neg a \wedge \neg b \quad h \leftarrow a\}$$

Example 6.7.2 For the BDD shown in Example 6.6.4 on the right, we obtain the following corresponding logic program:

$$\begin{aligned} P_{enc} := \{\ & o_1 \leftarrow i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge \neg i_5 \wedge \neg i_6 \wedge \neg i_7 \wedge \neg i_8 \\ & o_2 \leftarrow \neg i_1 \wedge i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge \neg i_5 \wedge \neg i_6 \wedge \neg i_7 \wedge \neg i_8 \\ & o_3 \leftarrow \neg i_1 \wedge \neg i_2 \wedge i_3 \wedge \neg i_4 \wedge \neg i_5 \wedge \neg i_6 \wedge \neg i_7 \wedge \neg i_8 \\ & o_4 \leftarrow \neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \quad \neg i_5 \wedge \neg i_6 \wedge \neg i_7 \wedge \neg i_8 \\ & o_5 \leftarrow \neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge i_5 \wedge \neg i_6 \wedge \neg i_7 \wedge \neg i_8 \\ & o_6 \leftarrow \neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge \neg i_5 \wedge i_6 \wedge \neg i_7 \wedge \neg i_8 \\ & o_7 \leftarrow \neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge \neg i_5 \wedge \neg i_6 \wedge i_7 \wedge \neg i_8 \\ & o_8 \leftarrow \neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge \neg i_5 \wedge \neg i_6 \wedge \neg i_7 \wedge i_8\ \} \end{aligned}$$

From Lemma 6.7.2 we can conclude that the resulting logic programs are also propositional representation for a given unit in the sense that for all activating interpretations I there is a clause in the program whose body is mapped to $\top$ under I. In this section, we closed the neural-symbolic cycle which started with a propositional program, embedding it into a connectionist system, training the network, and extracting the refined program using the CoOp-approach presented above. A brief evaluation of the extraction is presented in the following section.

6.8 Evaluation

The Prolog implementation to extract the BDD from a given perceptron was evaluated using perceptrons of different size. The results are shown in Table 6.1. For each number of inputs, 100 different randomised perceptrons have been tested and the average sizes were recorded. The table shows for 1 to 20 inputs: the number of (minimal) coalitions, the size of full search tree (FSTS), the size of the BDD (BDDS) and the resulting number of BDD-nodes per (minimal) coalition. The extraction using the full search tree is not feasible due to the exponential growth. Please note that the number of minimal coalitions is a very conservative lower bound for the size of the pruned search tree, because those trees have at least one node per minimal coalition. A better estimate would be twice the number of minimal coalitions, which holds if the tree was perfectly balanced. The results show that the use of binary decision diagrams as in the CoOp approach as presented above yields a very compact representation. Even though the number of nodes in the BDD grows, the ratio between nodes and the number of minimal coalitions decreases.

A second experiment has been performed to show the effect of the usage of integrity con-

# Inputs	# of (min.) coal.		FSTS	BDDS	node / (min.) coal.	
1	0.58	(0.58)	2	2.58	4.448	(4.448)
2	1.17	(0.81)	4	3.26	2.786	(4.024)
3	2.59	(1.55)	8	4.52	1.745	(2.916)
4	5.24	(2.21)	16	5.77	1.101	(2.610)
5	11.18	(4.31)	32	8.49	0.759	(1.969)
6	23.57	(7.17)	64	12.04	0.510	(1.679)
7	50.49	(12.65)	128	16.91	0.334	(1.336)
8	98.26	(21.71)	256	24.41	0.248	(1.124)
9	192.81	(34.89)	512	35.52	0.184	(1.018)
10	407.63	(63.51)	1024	49.97	0.122	(0.786)
11	828.66	(123.84)	2048	72.70	0.087	(0.587)
12	1597.41	(196.49)	4096	105.39	0.065	(0.536)
13	3392.85	(410.15)	8192	152.96	0.045	(0.372)
14	6736.62	(751.06)	16384	223.91	0.033	(0.298)
15	13721.20	(1270.45)	32768	313.25	0.022	(0.246)
16	27499.00	(2482.86)	65536	445.35	0.016	(0.179)
17	54097.60	(4447.65)	131072	633.63	0.011	(0.142)
18	109920.00	(7769.50)	262144	918.80	0.008	(0.118)
19	228097.00	(16220.20)	524288	1291.80	0.005	(0.079)
20	435197.00	(25681.70)	1048576	1863.90	0.004	(0.072)

Table 6.1: *Comparison of the search tree sizes and the size of the coalition BDDs: The table shows average numbers for different number of inputs (# Inputs), the number of (minimal) coalitions (# of (min.) coal.), the size of the full search tree (FSTS), the size of the coalition BDD (BDDS) and the number of BDD nodes per (minimal) coalition (node / (min.) coal.). All numbers were collected from 100 random perceptrons per size.*

straints while extracting the BDDs. An network with 6 inputs, 4 hidden and 2 output units has been used for the experiment. The possible inputs have been constrained by a $\max_n$ integrity constraint analog to Example 6.6.4, for $0 \leq n \leq 6$. The results are presented in Figure 6.12. For every n the experiment has been conducted for the same 100 randomised networks and the following numbers have been collected: the size of the sub-BDD encoding the constraint (ic), the minimal, maximal and average size of the final BDD. Please note that the numbers show the total number of internal nodes constructed for the BDD, i.e., including all necessary intermediate nodes. There are a number of immediate observation:

- For $n = 1$, i.e., the biggest restriction, we obtain very small BDDs.
- The size of the BDD grows up to $n = 4$ and decreases again for $n > 4$.

From those observations we can conclude:

- The incorporation of integrity constraints into the extraction process can lead to savings in terms of nodes constructed for the final BDD. Without their use during the extraction, we would have needed to construct the BDD corresponding to $n = 6$. This BDD of ≈ 400 nodes would have to be refined with respect to the constraints afterwards.
- There seem to be cases where the use of integrity constrain yields larger BDDs, but nonetheless, the final BDD does not have to be revised afterwards, and the difference is not too big.

		|BDD|		
n	**ic**	min	max	**aver.**
1	69	94	151	118.14
2	117	194	497	341.90
3	111	258	791	502.80
4	63	199	863	514.14
5	22	111	813	442.62
6	12	92	758	396.98

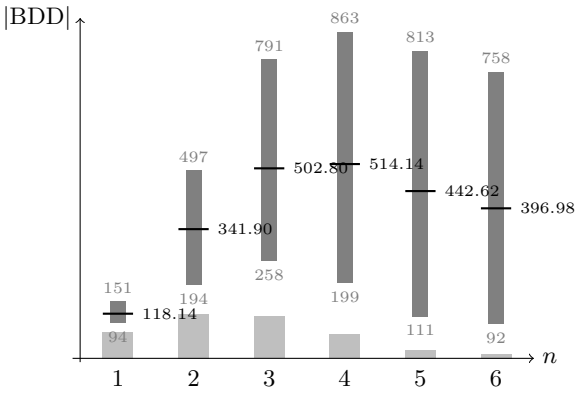


Figure 6.12: *Resulting BDD sizes of the extraction for different* $\max_n$*-integrity constraints.*

6.9 Summary

In this chapter a novel approach for the extraction of propositional rules from feed-forward networks of threshold units has been presented. After decomposing the network into perceptrons, binary decision diagrams representing preconditions that activate or inactivate the perceptron have been extracted. Those intermediate representations can be composed using the usual algorithms for BDDs, or they can be combined during their construction by extracting one into the other. This latter approach allows also the straight forward incorporation of integrity constraints.

The CoOp approach as presented here is applicable to all feed-forward networks composed of binary threshold units computing Θ_{-1}^{+1}. This limitation can be softened by allowing arbitrary symmetric threshold functions Θ_{-a}^{+a}. The symmetry is necessary to construct negative and

positive forms of the perceptron without changing the global network function. And for all Θ_{-1}^{+1}-networks we find the extraction to be sound and complete as shown in Theorem 6.5.5 and 6.6.2.

We have also discussed first experimental results indicating a good performance of the approach. Nonetheless those are not yet a detailed analysis and much work remains to be done. In particular, the extraction for non-threshold units has to be studied. In the encoder examples discussed above we simply applied the variant for threshold units to a network composed of sigmoidal units. Nonetheless, the result coincide with our expectations. This is due to the fact, that networks when trained to compute crisp decisions tend to behave like threshold networks. But the details of this need to be investigated in the future. Using the results from Chapter 3 up to this one, we obtain a full implementation of the neural-symbolic cycle as presented in Figure 1.1.

A detailed comparison with other (related) approaches is also subject to future research. But in order to perform real experiments, the full re-implementation is probably necessary, because the algorithms presented above have not been optimized for speed. After (i) extending the approach to non-threshold units and (ii) the re-implementing the algorithms, it would be desirable to finally close the neural-symbolic cycle and apply the complete system in a real-world application domain.

7 Embedding First-Order Rules into Connectionist Systems

... where we show how to embed first-order rules into connectionist systems. After a repetition of the feasibility result from [HKS99] in Section 7.1, we discuss the embedding of interpretations into vectors of real numbers. This is done in great detail because this embedding and its properties are of central importance. Afterwards, we show how to embed the associated consequence operator in principle and re-obtain the existential results on a more general level than in [HKS99]. In Section 7.4.1, we show how to approximate the embedded T_P-operator using piecewise constant functions. This is the basis for all connectionist representations discussed thereafter. Approximating sigmoidal, RBF and vector based networks are constructed in Section 7.4.2, 7.4.3 and 7.4.4, respectively. Then we discuss the training of the obtained vector-based networks and conclude with a summary in Section 7.7.

This chapter is partly based on [BHW05] and [BHHW07], and on many fruitful discussions with Andreas Witzel.

As already mentioned in Section 1.4, a central problem for the neural-symbolic integration is the representation of first-order rules within a connectionist setting. This would result in the computation or approximation of the associated semantic operator. Steffen Hölldobler, Yvonne Kalinke and Hans-Peter Störr showed in 1999 the existence of approximating networks for the consequence operators of acyclic logic programs [HKS99]. But this was shown with the help of Funahashi's theorem which is non-constructive. It was not clear how to construct such an approximating network until we showed this in [BHW05]. Unfortunately, it turned out that the accuracy is very limited if implemented on a real computer, because of the limited precision of floating point arithmetic. Therefore, we extended our approach in [BHHW07] such that it allows for arbitrary precision. All three results are discussed below. Before going into details we need to answer the following questions:

- Why do we need to approximate the T_P-operator?
- What does approximation mean in our context?

The first question is easily answered: Even a single application of the operator can lead to infinite results. Assume a program containing the fact $p(X)$. Applying the T_P-operator once (to an arbitrary interpretation), leads to a result containing infinitely many atoms, namely all $p(X)$-atoms for every $X \in U_{\mathcal{L}}$. In this simple example, we might be able to represent the result in a finite way, but it might become arbitrary complex for other programs. so-called rational models were developed to tackle this representational problem for certain programs [Bor96]. Unfortunately, there is no way to compute an upper bound on the size of this rational representation, and hence it does not give any immediate advantages here. Because we are not aware of any other finite representation, we will concentrate on the standard representation using Herbrand interpretations.

In principle, there are two ways to approximate a given T_P-operator. On the one hand, we can design an approximating function for a given accuracy. This leads to an increasing number of hidden layer units in the resulting networks. Therefore, we call it *approximation in space*. The approaches presented below follow this line. Alternatively, we could construct a system that approximates a single application of T_P the better the longer it runs. This *approximation in time* was used in [BH04] and [BHdG05] as described in Section 1.2.2. Networks constructed along the approximation in space approach are more or less standard architectures with many hidden layer units, while the others are usually very small but use non-standard architectures and units. Example 7.0.1 serves as a running example throughout this chapter.

7.1 Feasibility of the Core Method

As mentioned in Section 2.5, it is well known that multilayer feed-forward networks are universal approximators [HSW89, Fun89] for certain real functions. In particular, for all continuous functions on compact subsets. Hence, if we find a suitable way to represent first-order interpretations by (finite vectors of) real numbers, then feed-forward networks may be used to approximate the meaning function of such programs. It is necessary that such a representation is compatible with both, the logic programming and the neural network paradigm. We define a homeomorphic embedding from the space of interpretations into some (compact) subset of the real numbers. Following [HKS99], we use level mappings to realise this embedding. For some Herbrand interpretation I, some bijective level mapping $|\cdot|$ from $B_{\mathcal{L}}$ to $\mathbb{N}^+$ and $b > 2$, we define the *embedding function* $\iota : B_{\mathcal{L}} \to \mathbb{R}$ and its extension $\iota : \mathcal{I}_{\mathcal{L}} \to \mathbb{R}$ as follows:

$$\iota : B_{\mathcal{L}} \to \mathbb{R} : A \mapsto b^{-|A|} \qquad \iota : \mathcal{I}_{\mathcal{L}} \to \mathbb{R} : I \mapsto \sum_{A \in I} \iota(A). \tag{7.1}$$

Example 7.0.1 The following (first-order) logic program serves as a running example throughout this chapter:

$$\begin{array}{lll} P = \{ & e(0). & \%\ 0 \text{ is even} \\ & e(s(X)) \leftarrow o(X). & \%\ \text{the successor } s(X) \text{ of an odd } X \text{ is even} \\ & o(X) \leftarrow \neg e(X). \quad \} & \%\ X \text{ is odd if it is not even} \end{array}$$

The obviously intended model is $M = \{e(s^n(0)) \mid n \text{ is even}\} \cup \{o(s^n(0)) \mid n \text{ is odd}\}$, which is also the least fixed point of the associated T_P-operator. But note that $B_\mathcal{L} = \{e(s^n(0)) \mid n \in \mathbb{N}\} \cup \{o(s^n(0)) \mid n \in \mathbb{N}\}$ is also a model. A suitable level mapping $|\cdot| : B_\mathcal{L} \to \mathbb{N}^+$ would be:

$$e(s^n(0)) \mapsto 2n + 1 \text{ and}$$
$$o(s^n(0)) \mapsto 2n + 2,$$

i.e., all e-atoms are mapped to the odd numbers and the o-atoms are mapped to even numbers starting with 2. Using this level mapping we find that P is acyclic.

We use $\mathfrak{C}_b := \{\iota(I) \mid I \in \mathcal{I}_\mathcal{L}\} \subset \mathbb{R}$ to denote the set of all embedded interpretations. This embedding results in a "binary" representation[1] of a given interpretation I in the number system with base b. After embedding an interpretation and looking at its representation within this number system we find a 1 at each position $|A|$ for all $A \in I$. This is shown in Example 7.1.1.

Example 7.1.1 The following table shows the interpretations M and $B_\mathcal{L}$ from Example 7.0.1, their "binary" representation and the corresponding real value for $b = 4$. The ground atoms corresponding to the digits in the binary representation are shown in grey.

Interpretation	"Binary" Value	Real Value
$\{\}$	$0.0\ 0\ 0\ 0\ 0\ 0\ 0\ldots_4$	0.0_{10}
$\{e(0)\}$	$0.1\ 0\ 0\ 0\ 0\ 0\ 0\ldots_4$	0.25_{10}
$\{e(0), o(s(0)), e(s^2(0)), o(s^3(0)), \ldots\} = M$	$0.1\ 0\ 0\ 1\ 1\ 0\ldots_4$	$\approx 0.254_{10}$
$\{e(s^i(0)), o(s^i(0)) \mid 0 \leq i\} = B_\mathcal{L}$	$0.1\ 1\ 1\ 1\ 1\ 1\ldots_4$	$0.\bar{3}_{10}$

As mentioned above, we require the embedding to be homeomorphic, i.e., to be continuous, bijective and to have a continuous inverse. Being homeomorphic ensures that ι is at least a promising candidate to bridge the gap between logic programs and connectionist networks. We can now construct the *real valued version* of T_P as follows:

$$f_P : \mathfrak{C}_b \to \mathfrak{C}_b : x \mapsto \iota(T_P(\iota^{-1}(X)))$$

Because ι is a homeomorphism, we know that f_P is a correct embedding of T_P in the sense that all structural information is preserved. Furthermore, we know that ι is continuous what allows to conclude that f_P is continuous for a continuous T_P. Using Funahashi's result (Theo-

[1] We call the representation binary, because in any number system only the digits 0 and 1 are used.

rem 2.5.19 in Section 2.5), we can conclude that for those T_P-operators approximating networks exist.

Now we know that there must be approximating networks, but we do not know how to construct them yet. This is done in various ways in the following sections.

7.2 Embedding Interpretations into the Real Numbers

As shown in Example 7.1.1 we rely on a high accuracy while using the embedding presented in Equation (7.1). This high accuracy cannot be assumed while using the standard floating point numbers in computers. We could either use an infinite precision computer or represent an interpretation not as a single number but rather as a real vector. To allow for embeddings into vectors, we use multi-dimensional level mappings as introduced in Definition 2.3.12. A possible 2-dimensional mapping for our running example is given in Example 7.2.1.

Example 7.2.1 A possible 2-dimensional level mapping for P_2 is given as:

$$\|\cdot\| : B_{\mathcal{L}} \to (\mathbb{N}^+, \{1,2\}) \quad \text{with} \quad e(s^n(0)) \mapsto (n+1, 1) \quad \text{and} \quad o(s^n(0)) \mapsto (n+1, 2).$$

I.e., all *even*-atoms are mapped to the first dimension, and the *odd*-atoms to the second. This embedding applied to the interpretations from Example 7.1.1 yields the following values:

Interpretation	"Binary" Value $\begin{matrix}\lvert 0\rvert \\ \lvert s(0)\rvert \\ \lvert s(s(0))\rvert\end{matrix}$	Real Vector
$\{\}$	$\begin{pmatrix} 0.0\,0\,0\,0\,\ldots_4 \\ 0.0\,0\,0\,0\,\ldots_4 \end{pmatrix}$	$\begin{pmatrix} 0.00_{10} \\ 0.00_{10} \end{pmatrix}$
$\{e(0)\}$	$\begin{pmatrix} 0.1\,0\,0\,0\,\ldots_4 \\ 0.0\,0\,0\,0\,\ldots_4 \end{pmatrix}$	$\begin{pmatrix} 0.25_{10} \\ 0.00_{10} \end{pmatrix}$
$\{e(0), o(s(0)), e(s^2(0)), o(s^3(0)), \ldots\} = M$	$\begin{pmatrix} 0.1\,0\,1\,0\,\ldots_4 \\ 0.0\,1\,0\,\ldots_4 \end{pmatrix}$	$\begin{pmatrix} 0.26_{10} \\ 0.07_{10} \end{pmatrix}$
$\{e(s^i(0)), o(s^i(0)) \mid 0 \leq i\} = B_{\mathcal{L}}$	$\begin{pmatrix} 0.1\,1\,1\,1\,\ldots_4 \\ 0.1\,1\,1\,\ldots_4 \end{pmatrix}$	$\begin{pmatrix} 0.3\bar{3}_{10} \\ 0.3\bar{3}_{10} \end{pmatrix}$

Definition 7.2.1 (Multi-Dimensional Embedding) *Let $b \geq 3$, let $\|\cdot\| : B_{\mathcal{L}} \to (\mathbb{N}^+, \{1, \ldots, d\})$ be some d-dimensional level mapping and let $A \in B_{\mathcal{L}}$ be an atom with $\|A\| = (l_A, d_A)$. The* d-dimensional embedding $\boldsymbol{\iota} : B_{\mathcal{L}} \to \mathbb{R}^d$ *and its extensions $\boldsymbol{\iota} : \mathcal{I}_{\mathcal{L}} \to \mathbb{R}^d$ are defined as:*

$$\boldsymbol{\iota}(A) := (\iota^{[1]}(A), \ldots, \iota^{[d]}(A)) \quad \textit{with} \quad \iota^{[j]}(A) := \begin{cases} b^{-l_A} & \textit{if } j = d_A \\ 0 & \textit{else} \end{cases}$$

$$\boldsymbol{\iota}(I) := (\boldsymbol{\iota}^{[1]}(A), \ldots, \boldsymbol{\iota}^{[d]}(A)) \quad \textit{with} \quad \boldsymbol{\iota}^{[j]}(A) := \sum_{A \in I} \iota^{[j]}(A).$$

```
iota(_LM:1, _B, [], 0) :- !.
iota(LM:1, B, [H|I], X) :- !,
        level(LM:1, H, L:1),
        X1 is B^(-L),
        iota(LM:1, B, I, X2),
        X is X1+X2.
```

Figure 7.1: *Implementation of the embedding function ι for the 1-dimensional case: For a 1-dimensional level mapping* `LM : 1`*, a base* `B` *and an interpretation it computes the corresponding real value recursively by iterating over the elements of the interpretation.*

```
iota(LM:N, B, I, X) :- N > 1,
        iotaN(LM:N, B, I, X).

iotaN(_:N, _, [], X) :-
        vec0(N, X).
iotaN(LM, B, [A|I], X) :-
        iotaN(LM, B, I, X2),
        level(LM, A, L:D),
        X1 is B^(-L),
        vecElementAdd(X2, D, X1, X).
```

Figure 7.2: *Implementation for the embedding function $\boldsymbol{\iota}$ for multi-dimensional embeddings: As in Figure 7.1, the embedding of an interpretation is computed recursively.*

Example 7.2.2 Applying the embedding functions shown in Figure 7.1 and 7.2, we obtain the following encodings of the interpretations $I_1 = \{e(0)\}$ and $I_2 = \{e(0), o(s(0))\}$. `X11` is the 1-dimensional embedding of I_1, `X12` the 2-dimensional version, analogously for I_2.

```
?- iota(eo:1, 4, [e(0)], X11),
   iota(eo:1, 4, [e(0), o(s(0))], X12),
   iota(eo:2, 4, [e(0)], X21),
   iota(eo:2, 4, [e(0), o(s(0))], X22).

X11 = 0.25000
X12 = 0.25391
X21 = [0.25000, 0.00000]
X22 = [0.25000, 0.06250]
```

Figure 7.1 and 7.2 show the Prolog implementation of the embeddings ι and $\boldsymbol{\iota}$, respectively. Both compute the embedded value of a given interpretation with respect to a given base b and a given level mapping. Sample runs for some interpretations are shown in Example 7.2.2.

We assume the level mapping to be bijective and the space of all embedded interpretations wrt. the base b and dimension d is denoted by $\mathfrak{C}_b^d$. All approximation results presented below are based on this set with the usual maximum metric inherited from the real numbers.

Definition 7.2.2 (The Space $(\mathfrak{C}_b^d, m_\mathsf{m})$) *We define $\mathfrak{C}_b^d$ to be the set of all embedded interpretations, i.e.,*

$$\mathfrak{C}_b^d := \{\boldsymbol{\iota}(I) \mid I \in \mathcal{I}_\mathcal{L}\} \subset \mathbb{R}^d.$$

And we use $(\mathfrak{C}_b^d, m_\mathsf{m})$ to refer to this set equipped with the maximum metric m_m as introduced in Definition 2.2.3.

To proof that $\boldsymbol{\iota}$ is a homeomorphism, we need to show that it is bijective and continuous and that the inverse is continuous as well. Bijectivity allows to relate the space of interpretations and their embeddings as well as the T_P-operator and its embedded version in a precise way, because it ensures that there is a one-to-one correspondence between interpretations and members of $\mathfrak{C}_b^d$. We show the injectivity of $\boldsymbol{\iota} : \mathcal{I}_\mathcal{L} \to \mathbb{R}^d$, which obviously ensures bijectivity of $\boldsymbol{\iota} : \mathcal{I}_\mathcal{L} \to \mathfrak{C}_b^d$.

Lemma 7.2.3 *Let $\|\cdot\|$ be a bijective d-dimensional level mapping and let $\boldsymbol{\iota}$ be as defined above. Then, $\boldsymbol{\iota}$ is an injection from $\mathcal{I}_\mathcal{L}$ to $\mathbb{R}^d$.*

> **Proof** We show that $\boldsymbol{\iota}$ is injective, i.e., that $\boldsymbol{\iota}(I) = \boldsymbol{\iota}(J)$ implies $I = J$ by assuming the existence of $I \neq J$ such that $\boldsymbol{\iota}(I) = \boldsymbol{\iota}(J)$ and deriving a contradiction. From $I \neq J$ follows that there is some $A \in B_\mathcal{L}$ with $\|A\| = (l, d)$ on which both disagree. Without loss of generality, we assume $A \in I$ and $A \notin J$. Furthermore, we assume that there is no atom with level less than l on which I and J disagree. Let $K := \{B \in I \mid \|B\|_L < l\}$, i.e., K contains all atoms with smaller level than A; and I and J agree on K. Furthermore, let I' and J' denote the remaining parts of I and J respectively, i.e., $I' := I \setminus (\{A\} \cup K)$ and $J' := J \setminus K$. Looking at dimension d only, we find $\boldsymbol{\iota}^{[d]}(I) = \boldsymbol{\iota}^{[d]}(K) + b^{-l} + \boldsymbol{\iota}^{[d]}(I')$ and $\boldsymbol{\iota}^{[d]}(J) = \boldsymbol{\iota}^{[d]}(K) + \boldsymbol{\iota}^{[d]}(J')$. I.e., to derive a contradiction it suffices to show that $\boldsymbol{\iota}^{[d]}(J') < b^{-l}$, which follows from the fact that $\boldsymbol{\iota}^{[d]}(J')$ can be at most $\frac{b^{-l}}{b-1}$ which is smaller than b^{-l}. □

Please note, that $\boldsymbol{\iota}$ is not injective for $b = 2$, because $0.011\bar{1}_2 = 0.1_2$. By definition, $\boldsymbol{\iota}$ is not only injective for $b \geq 3$ but also surjective on $\mathfrak{C}_b^d$. Hence, it is a bijection between $\mathcal{I}_\mathcal{L}$ and $\mathfrak{C}_b^d$, i.e., there is a unique $x \in \mathfrak{C}_b^d$ for each $I \in \mathcal{I}_\mathcal{L}$ and vice versa. In the sequel, we treat $\boldsymbol{\iota}$ as a function from $\mathcal{I}_\mathcal{L}$ to $\mathfrak{C}_b^d$.

Corollary 7.2.4 *$\boldsymbol{\iota}$ is a bijection between $\mathcal{I}_\mathcal{L}$ and $\mathfrak{C}_b^d$.*

The bijectivity of $\boldsymbol{\iota}$ ensures that there is a unique real representation for each interpretation. And that for each $x \in \mathfrak{C}_b^d$ there is a unique interpretation. But we would also like to relate distances on $\mathcal{I}_\mathcal{L}$ to distances in $\mathfrak{C}_b^d$. This relation is established in the following lemma. It shows that both spaces, $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ and $(\mathfrak{C}_b^d, m_\mathsf{m})$ are equivalent metric spaces as introduced in Definition 2.2.4. I.e., there are two constants e_1 and e_2 such that $e_1 \cdot m_{\mathcal{L},b}(I, J) \leq m_\mathsf{m}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) \leq e_2 \cdot m_{\mathcal{L},b}(I, J)$ holds for all $I, J \in \mathcal{I}_\mathcal{L}$.

Lemma 7.2.5 *$(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ and $(\mathfrak{C}_b^d, m_\mathsf{m})$ are equivalent metric spaces as introduced in Definition 2.2.4 with $e_1 = \frac{b-2}{b-1}$ and $e_2 = \frac{b}{b-1}$ and we find for all $I, J \in \mathcal{I}_\mathcal{L}$ that the following holds:*

$$e_1 \cdot m_{\mathcal{L},b}(I,J) \leq m_\mathsf{m}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) \leq e_2 \cdot m_{\mathcal{L},b}(I,J)$$

Proof As above and without loss of generality, we assume that I and J disagree on some atom A with $\|A\| = (l,d)$ with $A \in I$ and $A \notin J$, and agree on all atoms of smaller level. I.e., we find $m_{\mathcal{L},b}(I,J) = b^{-l}$ and

$$m_\mathsf{m}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) = \boldsymbol{\iota}^{[d]}(I) - \boldsymbol{\iota}^{[d]}(J)$$

Obviously, the difference between $\boldsymbol{\iota}^{[d]}(I)$ and $\boldsymbol{\iota}^{[d]}(J)$ is biggest, if I contains all atoms with level $> l$ and dimension d and J contains none of them. I.e., we obtain

$$\boldsymbol{\iota}^{[d]}(I) - \boldsymbol{\iota}^{[d]}(J) \leq b^{-l} + \sum_{i>l} b^{-i} = b^{-l} + \frac{1}{b-1} \cdot b^{-l} = \frac{b}{b-1} \cdot b^{-l}$$

We find the difference between $\boldsymbol{\iota}^{[d]}(I)$ and $\boldsymbol{\iota}^{[d]}(J)$ to be as small as possible, if J contains all and I contains none of them. I.e., we obtain

$$\boldsymbol{\iota}^{[d]}(I) - \boldsymbol{\iota}^{[d]}(J) \geq b^{-l} - \sum_{i>l} b^{-i} = b^{-l} - \frac{1}{b-1} \cdot b^{-l} = \frac{b-2}{b-1} \cdot b^{-l}$$

Therefore, we can conclude that for all $I, J \in \mathcal{I}_\mathcal{L}$ with $m_{\mathcal{L},b}(I,J) = 2^{-l}$ we find

$$\frac{b-2}{b-1} \cdot b^{-l} = \frac{b-2}{b-1} \cdot m_{\mathcal{L},b}(I,J) \leq m_\mathsf{m}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) \leq \frac{b}{b-1} \cdot m_{\mathcal{L},b}(I,J) = \frac{b}{b-1} \cdot b^{-l}.$$

□

From the fact that $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ and $(\mathfrak{C}_b^d, m_\mathsf{m})$ are equivalent with respect to $\boldsymbol{\iota} : \mathcal{I}_\mathcal{L} \to \mathfrak{C}_b^d$, we can conclude that $\boldsymbol{\iota}$ as well as $\boldsymbol{\iota}^{-1}$ are both continuous. This follows from the fact that we can compute a suitable δ for each $\varepsilon > 0$ such that $m_{\mathcal{L},b}(I,J) < \delta$ implies $m_\mathsf{m}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) < \varepsilon$. And we can also compute a δ' for each $\varepsilon' > 0$ such that $m_\mathsf{m}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) < \delta'$ implies $m_{\mathcal{L},b}(I,J) < \varepsilon'$.

Corollary 7.2.6 *$\boldsymbol{\iota} : B_\mathcal{L} \to \mathfrak{C}_b^d$ is a continuous mapping from $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ to $(\mathfrak{C}_b^d, m_\mathsf{m})$.*

Corollary 7.2.7 *$\boldsymbol{\iota}^{-1} : \mathfrak{C}_b^d \to B_\mathcal{L}$ is a continuous mapping from $(\mathfrak{C}_b^d, m_\mathsf{m})$ to $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$.*

In other words, Corollary 7.2.4 to 7.2.7 state that $\boldsymbol{\iota}$ is a *homeomorphism*, i.e., a structure-preserving mapping between the two metric spaces $(\mathfrak{C}_b^d, m_\mathsf{m})$ and $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$. Homeomorphisms play an important role in the theory of metric spaces, because central notions such as compactness and continuity are preserved [Wil70].

Corollary 7.2.8 *$\boldsymbol{\iota}$ is a homeomorphism between $(\mathbb{R}^d, m_\mathsf{m})$ and $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$.*

The compactness of $(\mathfrak{C}_b^d, m_\mathsf{m})$ follows from the compactness of $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ as shown in Proposition 2.3.17. Knowing that it is compact paves the way to the application of Funahashi's theorem.

Corollary 7.2.9 *$(\mathfrak{C}_b^d, m_\mathsf{m})$ is compact.*

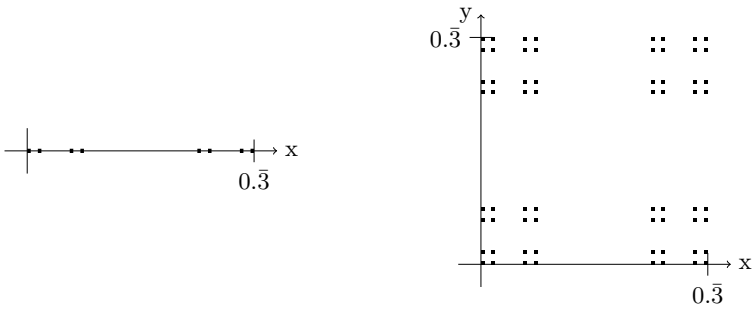


Figure 7.3: *The set of embedded interpretations $\mathfrak{C}_4^1$ on the left and $\mathfrak{C}_4^2$ on the right*

Figure 7.3 shows the plots of $\mathfrak{C}_4^1$ and $\mathfrak{C}_4^2$. Both show a self-similar structure as known from iterated function systems. I.e., after zooming into the graph, the result looks like the whole plot. And indeed this is shown formally in the sequel. First, we define a suitable iterated function system and afterwards we show that its attractor coincides with the set of embedded interpretations.

Definition 7.2.10 ($\mathfrak{C}_b^d$-IFS) *Let $d \geq 1$, $b > 2$, $\omega_\bot : x \mapsto \frac{x}{b}$ and $\omega_\top : x \mapsto \frac{x}{b} + \frac{1}{b}$. We define the iterated function system $\mathfrak{C}_b^d$-IFS as follows:*

$$\mathfrak{C}_b^d\text{-IFS} := \left\langle \left(\mathbb{R}^d, m_{\mathfrak{E}}\right), \Omega \right\rangle \text{ with } \Omega = \{\omega_S \mid S \subseteq \{1,\ldots,d\}\} \quad \text{and}$$
$$\omega_S : \mathbb{R}^d \to \mathbb{R}^d : \left(x^{[1]},\ldots,x^{[d]}\right) \mapsto \left(y^{[1]},\ldots,y^{[d]}\right) \text{ with } y^{[i]} = \begin{cases} \omega_1(x_\bot) & \text{if } i \notin S \\ \omega_2(x_\top) & \text{if } i \in S \end{cases}$$

The $\mathfrak{C}_b^d$-IFS just introduced are obviously IFSs as introduced in Definition 2.2.17, because $\omega_\bot$ and $\omega_\top$ are contractions and the underlying space is complete. They contain all possible mappings on $\mathbb{R}^d$ whose single dimensions are computed by either $\omega_\bot$ or $\omega_\top$. The set S attached to the mappings $\omega_S \in \Omega$ contains all those dimensions in which $\omega_\top$ is used, while all dimensions not contained in S are computed using $\omega_\bot$. Before showing that the attractor of $\mathfrak{C}_b^d$-IFS actually coincides with $\mathfrak{C}_b^d$, we introduce some further notation. The intermediate steps of the construction of the attractor while starting from one particular interval play an important role in the sequel. This starting interval – the *base hyper-cube* $\mathfrak{U}_b^d$ – is the smallest hyper cube containing $\mathfrak{C}_b^d$. The construction of $\mathfrak{C}_b^1$ and $\mathfrak{C}_b^2$ starting with the corresponding base hyper-cubes are shown in Example 7.2.3 and 7.2.4. We use $\Omega^n\left(\mathfrak{U}_b^d\right)$ to dentote the set containing all images of $\mathfrak{U}_b^d$ after n applications of some mapping from Ω. Obviously, $\Omega^n\left(\mathfrak{U}_b^d\right)$ contains $(2^d)^n$ images of $\mathfrak{U}_b^d$, each scaled by b^{-n}.

Definition 7.2.11 (Base Hyper-Cube $\mathfrak{U}_b^d$) *Let $\mathfrak{C}_b^d\text{-IFS} = \left\langle (\mathbb{R}^d, m_{\mathfrak{E}}), \Omega \right\rangle$ be as defined above for some $\mathfrak{C}_b^d$, $d \geq 1$ and $b \geq 3$. Then we define the* base hyper-cube $\mathfrak{U}_b^d$ wrt. b and d *as:*

$$\mathfrak{U}_b^d = \left[0, \frac{1}{b-1}\right]^d = \left\{ \left(x^{[1]},\ldots,x^{[d]}\right) \in \mathbb{R}^d \mid 0 \leq x^{[i]} \leq \frac{1}{b-1} \right\}$$

The set $\Omega^n\left(\mathfrak{U}_b^d\right)$ of all base hyper-cubes of level n *is defined recursively as follows:*

$$\Omega^0\left(\mathfrak{U}_b^d\right) := \left\{\mathfrak{U}_b^d\right\}$$
$$\Omega^{n+1}\left(\mathfrak{U}_b^d\right) := \left\{\omega(u) \mid u \in \Omega^n\left(\mathfrak{U}_b^d\right) \text{ and } \omega \in \Omega\right\}$$

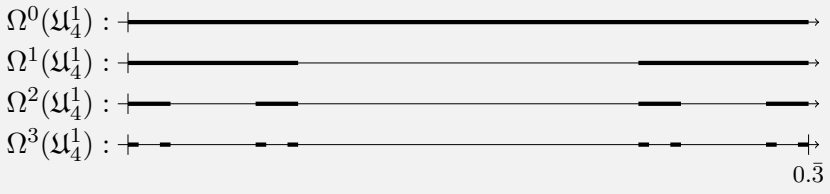
We furthermore define the union of all hyper-cubes of some level n as

$$(\mathfrak{U}_b^d)^n := \bigcup_{u \in \Omega^n\left(\mathfrak{U}_b^d\right)} u$$

Example 7.2.3 For $d = 1$ and $b = 4$, we find $\mathfrak{U}_4^1 = \left[0, \frac{1}{3}\right]$ and $\mathfrak{C}_b^1$-IFS $:= \left\langle (\mathbb{R}, m_{\mathfrak{E}}), \{\omega_{\{\}}, \omega_{\{1\}}\} \right\rangle$ with

$$\omega_{\{\}} : x \mapsto \left(\frac{x}{4}\right), \qquad \omega_{\{1\}} : x \mapsto \left(\frac{x}{4} + \frac{1}{4}\right).$$

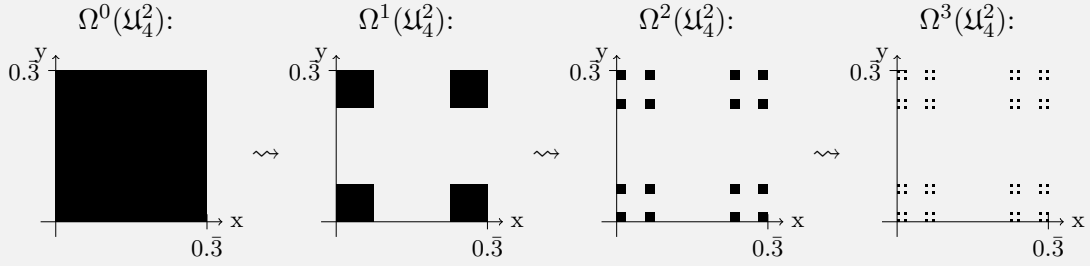
The first four iterations of the construction of $\mathfrak{C}_4^1$ starting at $\mathfrak{U}_4^1$:



Example 7.2.4 For $d = 2$ and $b = 4$, we find $\mathfrak{U}_4^2 = \left[0, \frac{1}{3}\right]^2$ and $\mathfrak{C}_b^2$-IFS $:= \left\langle (\mathbb{R}^2, m_{\mathfrak{E}}), \Omega \right\rangle$ with $\Omega = \{\omega_{\{\}}, \omega_{\{1\}}, \omega_{\{2\}}, \omega_{\{1,2\}}\}$ and

$$\omega_{\{\}} : (x, y) \mapsto \left(\frac{x}{4}, \frac{x}{4}\right), \qquad \omega_{\{1\}} : (x, y) \mapsto \left(\frac{x}{4} + \frac{1}{4}, \frac{x}{4}\right),$$
$$\omega_{\{2\}} : (x, y) \mapsto \left(\frac{x}{4}, \frac{x}{4} + \frac{1}{4}\right), \qquad \omega_{\{1,2\}} : (x, y) \mapsto \left(\frac{x}{4} + \frac{1}{4}, \frac{x}{4} + \frac{1}{4}\right).$$

The first four iterations of the construction of $\mathfrak{C}_4^2$ starting at $\mathfrak{U}_4^2$:



Lemma 7.2.12 *Let $b \geq 3$, $d \geq 1$, $n \geq 0$, $\Omega^n\left(\mathfrak{U}_b^d\right)$ and $\mathfrak{C}_b^d$ be as defined above. Then we find*

$$\mathfrak{C}_b^d \subset \bigcup \Omega^n\left(\mathfrak{U}_b^d\right)$$

Proof We show this by induction on n.

Induction basis $(n = 0)$ By definition we find $\bigcup \Omega^n\left(\mathfrak{U}_b^d\right) = \mathfrak{U}_b^d$ and as an immediate consequence $\mathfrak{C}_b^d \subset \bigcup \Omega^0\left(\mathfrak{U}_b^d\right)$.

Induction hypothesis $\mathfrak{C}_b^d \subset \bigcup \Omega^n\left(\mathfrak{U}_b^d\right)$, i.e., for all $x \in \mathfrak{C}_b^d$ exists a $u \in \Omega^n\left(\mathfrak{U}_b^d\right)$ with $x \in u$.

Induction step $(n \rightsquigarrow n+1)$ For $x \in \mathfrak{C}_b^d$ we find some $I \in \mathcal{I}_\mathcal{L}$, such that $\boldsymbol{\iota}(I) = x$. Let $I' = \{A \mid A \in I \text{ and } \|A\|_L > 1\}$ and let $I'' = \{A \in I \mid \|A\|_L = 1\}$. We construct the *shifted* interpretation $J = \{B \mid A \in I' \text{ with } \|A\| = (l_A, d) \text{ and } \|B\| = (l_A + 1, d)\}$. For $v = \boldsymbol{\iota}(J)$ we find $\frac{v}{b} = \boldsymbol{\iota}(I')$ and for $\omega_S \in \Omega$ with $S = \{\|A\|_D \mid A \in I''\}$, we find

$$\begin{aligned}
\omega_S(v) &= \omega_S(v_1, \ldots, v_d) \\
&= (y'_1, \ldots, y'_d) \quad \text{with} \quad y'_i = \begin{cases} \frac{v_i}{b} & \text{if } i \notin S \\ \frac{v_i}{b} + \frac{1}{b} & \text{if } i \in S \end{cases} \\
&= \frac{v}{b} + \left(z'_1, \ldots, z'_d\right) \quad \text{with} \quad z'_i = \begin{cases} 0 & \text{if } i \notin S \\ \frac{1}{b} & \text{if } i \in S \end{cases} \\
&= \frac{v}{b} + \boldsymbol{\iota}(I'') = \boldsymbol{\iota}(I') + \boldsymbol{\iota}(I'') = \boldsymbol{\iota}(I' \cup I'') = \boldsymbol{\iota}(I) = x
\end{aligned}$$

From the induction hypothesis we know that there is a $u \in \Omega^n\left(\mathfrak{U}_b^d\right)$ with $v \in u$. Using ω_S we find $\omega_S(u) \in \Omega^{n+1}\left(\mathfrak{U}_b^d\right)$ and obviously $x \in \omega_S(u)$. □

The following lemma links $\mathfrak{C}_b^d$ and the corresponding $\mathfrak{C}_b^d$-IFS in the desired way. Namely by stating that the attractor of the IFS coincides with $\mathfrak{C}_b^d$.

Lemma 7.2.13 *For $d \geq 1$, $b \geq 3$, $\mathfrak{C}_b^d$ and $\mathfrak{C}_b^d$-IFS as defined above, we find $\mathfrak{C}_b^d = \mathcal{A}(\mathfrak{C}_b^d\text{-IFS})$.*

Proof $\mathfrak{C}_b^d \subseteq \Omega(\mathfrak{C}_b^d)$ follows immediately from Lemma 7.2.12, because the J constructed in the proof above is an interpretation and hence $v \in \mathfrak{C}_b^d$. $\mathfrak{C}_b^d \supseteq \Omega(\mathfrak{C}_b^d)$ can be shown analogously by reversing the shifts. □

For each $u \in \Omega^n\left(\mathfrak{U}_b^d\right)$ there is a sequence of ω_{S_i} such that $\omega_{S_1}(\omega_{S_2}(\ldots\omega_{S_n}(\mathfrak{U}_b^d))) = \omega_u(\mathfrak{U}_b^d) = u$. Therefore, we can characterise a single hyper-cube u by this mapping ω_u. But we can also characterise it by some particular interpretation I such that $\boldsymbol{\iota}(I) \in u$. We use the smallest interpretation I_u, i.e., the I_u whose embedding is the point in u that is closest to $\mathbf{0}$ and we find $\boldsymbol{\iota}(I_u) = \omega_u(\mathbf{0})$.

Definition 7.2.14 (ω_u, I_u) *Let $\mathfrak{C}_b^d$, $\Omega^n\left(\mathfrak{U}_b^d\right)$ and $\mathfrak{C}_b^d$-IFS $= \left\langle (\mathbb{R}^d, m_\mathfrak{E}), \Omega \right\rangle$ be as defined above for some $d \geq 1$ and $n \geq 0$. For each $u \in \Omega^n\left(\mathfrak{U}_b^d\right)$, $\omega_{S_1}, \ldots, \omega_{S_n}$ with $\omega_{S_1}(\omega_{S_2}(\ldots\omega_{S_n}(\mathfrak{U}_b^d))) = u$ and $\omega_{S_i} \in \Omega$, we define $\omega_u : \mathbb{R}^d \to \mathbb{R}^d$ and $I_u \in \mathcal{I}_\mathcal{L}$ as follows:*

$$\omega_u : x \mapsto \omega_{S_1}(\omega_{S_2}(\ldots\omega_{S_n}(x))) \qquad I_u = \{A \mid A \in B_\mathcal{L}, \|A\| = (l, d) \text{ and } d \in S_l\}$$

Lemma 7.2.15 *Let ω_u and I_u be as defined above for $u \in \Omega^n\left(\mathfrak{U}_b^d\right)$. Then we find $\omega_u(\mathbf{0}) = \boldsymbol{\iota}(I_u)$.*

Proof By definition, we find $\boldsymbol{\iota}(I_u) \in u$. Furthermore, we know that I_u does not contain an atom with level $> n$. Therefore, we can conclude, that $\omega_u(\mathbf{0}) = \boldsymbol{\iota}(I_u)$. □

Using I_u just introduced, we are able to define the inverse κ_n for our embedding function $\boldsymbol{\iota}$. This is defined with respect to some level n and returns I_u for all $x \in u \in \Omega^n\left(\mathfrak{U}_b^d\right)$. I.e., κ_n can be used to map real vectors from $\Omega^n\left(\mathfrak{U}_b^d\right) \supset \mathfrak{C}_b^d$ to interpretations.

Definition 7.2.16 *Let $\Omega^n\left(\mathfrak{U}_b^d\right)$, $(\mathfrak{U}_b^d)^n$ and I_u be as defined above for some $n \geq 0$. Then we define κ_n as follows:*

$$\kappa_n : (\mathfrak{U}_b^d)^n \to \mathcal{I}_\mathcal{L} : x \mapsto I_u \text{ with } x \in u \text{ and } u \in \Omega^n\left(\mathfrak{U}_b^d\right)$$

The following proposition links $\boldsymbol{\iota}$ and κ_n in the expected way, by showing that κ_n is indeed an inverse for $\boldsymbol{\iota}$ – at least up to some level n.

Proposition 7.2.17 *Let $n \geq 0$ and let $\Omega^n\left(\mathfrak{U}_b^d\right)$, I_u and κ_n be as defined above. Let $x \in \mathfrak{C}_b^d$ and $y \in \bigcup \Omega^n\left(\mathfrak{U}_b^d\right)$ such that $m_{\mathsf{m}}(x,y) \leq \frac{b^{-n}}{b-1}$. Then we find $m_{\mathcal{L},b}\left(\boldsymbol{\iota}^{-1}(x), \kappa_n(y)\right) \leq b^{-n}$.*

> **Proof** From $x \in \mathfrak{C}_b^d$, we can conclude that there is a $u \in \Omega^n\left(\mathfrak{U}_b^d\right)$ such that $x \in u$ and that $m_{\mathcal{L},b}(I_u, \boldsymbol{\iota}^{-1}(x)) \leq b^{-n}$. Furthermore, from $m_{\mathsf{m}}(x,y) \leq \frac{b^{-n}}{b-1}$ follows that $y \in u$. Hence we obtain $m_{\mathcal{L},b}\left(\boldsymbol{\iota}^{-1}(x), \kappa_n(y)\right) \leq b^{-n}$. □

Corollary 7.2.18 *For $n \geq 0$, κ_n as defined above and $x \in \mathfrak{C}_b^d$ we find*

$$m_{\mathcal{L},b}\left(\boldsymbol{\iota}^{-1}(x), \kappa_n(x)\right) \leq b^{-n}.$$

In this section, we discussed how to embed interpretations into real vectors. This was done using the multi-dimensional level mapping. We furthermore showed that the resulting space of all embedded interpretations together with the maximum metric is compact and gave a suitable iterated function system generating it. Finally, we defined an inverse of the embedding which uses the first n steps of the construction of the attractor. This mapping κ_n allows to extract interpretations from real vectors, which are not necessarily embedded interpretations.

In the following section we show how to obtain an embedded version of the T_P-operator, which can be approximated using connectionist systems.

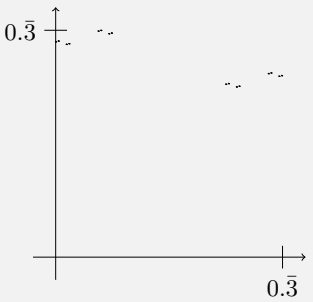
7.3 Embedding the Consequence Operator into the Real Numbers

After discussing the link between interpretations and real numbers in the previous section, we concentrate on the semantic operator T_P now. We show how to embed it into the reals and we show that this embedding is indeed suitable for our approach.

Definition 7.3.1 *Let $\boldsymbol{\iota}$ be a d-dimensional embedding as introduced above. The corresponding d-*dimensional embedding *f_P of a given T_P-operator is defined as follow:*

$$f_P : \mathfrak{C}_b^d \to \mathfrak{C}_b^d : \mathbf{x} \mapsto \boldsymbol{\iota}\left(T_P\left(\boldsymbol{\iota}^{-1}(\mathbf{x})\right)\right).$$

Example 7.3.1 The plot of f_P for our running example:



Please note that not only the domain $\mathfrak{C}_b^d$, but also the plot from Example 7.3.1 shows a self-similar structure. As already discussed in Section 1.2.2, this discovery led to a completely different encoding of first-order rules within connectionist systems [Bad03, BH04].

From the fact that $\boldsymbol{\iota}$ is a homeomorphism, we can conclude that $T_P(I)$ can be computed using f_P. I.e., f_P is a sound encoding of T_P into the real numbers.

Corollary 7.3.2 *Let T_P be a consequence operator defined over $B_\mathcal{L}$, $\mathcal{I}_\mathcal{L}$ be the corresponding space of interpretations and let $\boldsymbol{\iota} : B_\mathcal{L} \to \mathfrak{C}_b^d$ be the bijective d-dimensional embedding as introduced in Definition 2.3.12. Then $T_P(I) = \boldsymbol{\iota}^{-1}(f_P(\boldsymbol{\iota}(I)))$.*

Again by using the fact that $\boldsymbol{\iota}$ is homeomorphic, we know that f_P is continuous if and only if T_P is continuous. This follows from the fact that the composition of two continuous functions is continuous itself.

Corollary 7.3.3 *T_P is continuous if and only if f_P is continuous.*

This, together with the compactness of $\mathfrak{C}_b^d$, allows to apply Funahashi's theorem and we can conclude that there is an approximating network for continuous T_P-operators. For acyclic programs and a 1-dimensional embeddings this was first shown as Theorem 3 in [HKS99].

Corollary 7.3.4 *Let P be a program with continuous T_P-operator. Then there exist a three layer network approximating T_P.*

But using Lemma 2.2.14 from Section 2.2, we can relate T_P and f_P in a even more precise way. Not only structural information is preserved under $\boldsymbol{\iota}$, but also the distances as shown in Lemma 7.2.5. From T_P being Lipschitz continuous on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ we can conclude that f_P is Lipschitz continuous on $(\mathfrak{C}_b^d, m_\mathsf{m})$.

Lemma 7.3.5 *Let T_P be Lipschitz continuous on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ with Lipschitz constant L_T, let $b \geq 3$ and let f_P be as defined above. Then f_P is Lipschitz continuous with constant $L_f = \frac{b}{b-2} \cdot L_T$.*

Proof Using $e_1 = \frac{b-2}{b-1}$ and $e_2 = \frac{b}{b-1}$ and the fact that $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ and $(\mathfrak{C}_b^d, m_\mathsf{m})$ are equivalent wrt. e_1, e_2 and $\boldsymbol{\iota}$ (Lemma 7.2.5) we can conclude that f_P is Lipschitz continuous with Lipschitz constant L_f and

$$L_f = \frac{e_2}{e_1} \cdot L_T = \frac{b}{b-1} \cdot \frac{b-1}{b-2} \cdot L_T = \frac{b}{b-2} \cdot L_T$$

The first equality $L_f = \frac{e_2}{e_1} \cdot L_T$ follows from Lemma 2.2.14. □

Building on the homeomorphism $\boldsymbol{\iota}$ presented in the previous section, we showed here how to embed the semantic operator T_P into the real numbers. It turned out that $\boldsymbol{\iota}$ can not only be used to link interpretations and real numbers, but is also suitable for the embedding of T_P because all structural information as well as distances are preserved. This allows to conclude the existence of approximating networks using Funahashi's result in Corollary 7.3.4, which was based on the observation that the embedding is continuous. But we could also show that Lipschitz continuity is preserved, which is a stronger notion, because it also allows to conclude that contractivity is preserved. We discuss this issue further in Section 7.5. In the following section we show how to approximate the embedded T_P-operator using connectionist systems.

7.4 Approximating the Embedded Operator using Connectionist Systems

We continue by showing how to approximate the embedded T_P-operator in a connectionist system. First, we develop an approximation by some piecewise constant function, which serves as the basis for the connectionist implementations. Afterwards, we show how to construct sigmoidal, RBF and vector-based networks approximating this function.

7.4.1 Approximation by Piecewise Constant Functions

As a first step towards a connectionist implementation of the T_P-operator we show how to approximate the embedded version f_P using piecewise constant functions. As described below, this is possible for all covered programs, i.e., for all programs without local variables.

Before giving the formal proofs, we recall some ideas from Section 2.3. There, we showed that we can approximate the T_P-operator of a logic program P up to some level n by some subprogram $\lfloor P \rceil_n$ of ground(P). I.e., we find $d(T_P(I), T_{\lfloor P \rceil_n}(I)) \leq 2^{-n}$ for all $I \in \mathcal{I}_\mathcal{L}$. Furthermore, we showed that $\lfloor P \rceil_n$ is finite if P is covered, i.e., if there are no local variables in P. We use $\hat{n}$ to denote the greatest relevant input level, i.e., the greatest level occurring in some body of $\lfloor P \rceil_n$. We do not need to consider atoms with level greater then $\hat{n}$ while approximating T_P up to level n. As mentioned above, we assume the level mapping to be bijective.

Definition 7.4.1 (Greatest Relevant (Input) Level $\hat{n}$) *Let P be a covered logic program over $B_\mathcal{L}$, $\|\cdot\|$ be some d-dimensional level mapping and $n \geq 0$. We define the* greatest relevant input level $\hat{n}$ *with respect to P and n as follows:*

$$\hat{n} = \max\{n_c \mid H \leftarrow L_1 \wedge \ldots \wedge L_c \in \lfloor P \rceil_n \text{ and } n_c = \max_{L \in \{L_1, \ldots, L_c\}} \|L\|_L\}$$

The greatest relevant level $\tilde{n}$ *is defined as*

$$\tilde{n} = \max(n, \hat{n})$$

Please recall that acyclic programs are those for which all atoms occurring in a body have a level less than the level of the head. Hence we find for some n that the greatest relevant input level is at most $n-1$ and that the greatest relevant level is n.

Corollary 7.4.2 *Let P be an acyclic logic program and $n \geq 0$. Then we find $\hat{n} \leq n-1$ and $\tilde{n} = n$.*

Please note that for acyclic programs P we find $T_{\lfloor P \rceil_n}$ and $f_{\lfloor P \rceil_n}$ to be piecewise constant functions. $T_{\lfloor P \rceil_n}$ is piecewise constant, because the underlying $\lfloor P \rceil_n$ consists of atoms up to level n only, i.e., $T_{\lfloor P \rceil_n}$ yields the same result if applied to interpretations which agree on all atoms up to level n. Therefore, we find $f_{\lfloor P \rceil_n}$ to be constant on all hyper squares of level $\hat{n}$. Below, we define another piecewise constant function $\mathfrak{P}^d_{b,n,P} : \mathfrak{C}^d_b \to \mathfrak{C}^d_b$ for some level n. The intervals on which it is constant are base hyper-cubes of level n. To each of those intervals we associate an output value $v \in \mathfrak{C}^d_b$. This value v is computed such that it is the centre of the hyper cube in which the actual output of $f_{\lfloor P \rceil_n}$ is. Let v be the associated value for some hyper cube u. Then the output of the function $\mathfrak{P}^d_{b,n,P}$ is v for all $x \in u$.

Definition 7.4.3 *Let $d \geq 1$, $b > 2$, let $\mathfrak{U}^d_b$ be the corresponding base hyper cube and P be a covered logic program. Furthermore, let $\Omega^n\left(\mathfrak{U}^d_b\right)$, ω_u and I_u be as in Definition 7.2.10 and 7.2.14, respectively. Then we define for each level $n \geq 0$ the set of constant pieces $\mathfrak{P}^d_{b,n}(P)$ as follows:*

$$\mathfrak{P}^d_{b,n}(P) := \left\{ \left(u, \boldsymbol{\iota}(T_{\lfloor P \rceil_n}(I_u)) + \left[\frac{b^{-n}}{b-1} \cdot \frac{1}{2} \right]^d \right) \mid u \in \Omega^n\left(\mathfrak{U}^d_b\right) \right\}.$$

We define the corresponding approximating piecewise constant function:

$$\mathfrak{P}^d_{b,n,P} : (\mathfrak{U}^d_b)^n \to \mathbb{R}^d : x \mapsto v \text{ for } (u, v) \in \mathfrak{P}^d_{b,n}(P) \text{ and } x \in u.$$

Finally we define the maximal approximation error ε_n as follows

$$\varepsilon_n = \frac{b^{-n}}{b-1} \cdot \frac{1}{2}$$

As mentioned above this piecewise constant function is the basis for all connectionist approximations presented below. If some function f approximates $\mathfrak{P}^d_{b,n,P}$ up to ε_n, we find that $\kappa_n(f)$ approximates $T_{\lfloor P \rfloor_n}$ up to level n.

Proposition 7.4.4 *Let P be some covered logic program, let $n \geq 0$ and let $\tilde{n}$ as well as $\mathfrak{P}^d_{b,\tilde{n},P}$ be defined as above. Furthermore, let $f : \mathfrak{C}^d_b \to \mathbb{R}^d$ be some approximation of $\mathfrak{P}^d_{b,\tilde{n},P}$ on $\mathfrak{C}^d_b$ up to $\varepsilon_{\tilde{n}}$. Then we find*

$$m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(f(\boldsymbol{\iota}(I)))) \leq b^{-n} \quad \text{for all } I \in \mathcal{I}_{\mathcal{L}}.$$

Proof From the fact that $m_{\mathsf{m}}\left(f(x), \mathfrak{P}^d_{b,\tilde{n},P}(x)\right) \leq \varepsilon_{\tilde{n}}$ and the way $\mathfrak{P}^d_{b,\tilde{n},P}$ is constructed, we can conclude that there is some $u \in \Omega^{\tilde{n}}\left(\mathfrak{U}^d_b\right)$ such that $f(x), \mathfrak{P}^d_{b,\tilde{n},P}(x) \in u$. I.e., we find $\kappa_n(f(x)) = \kappa_n(\mathfrak{P}^d_{b,\tilde{n},P}(x)) = \kappa_n(f_{\lfloor P \rfloor_n}(x))$. Using Corollary 7.3.2 and Proposition 7.2.17, we obtain

$$\begin{aligned} m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(f(\boldsymbol{\iota}(I)))) &= m_{\mathcal{L},b}(\boldsymbol{\iota}^{-1}(f_{\lfloor P \rfloor_n}(\boldsymbol{\iota}(I))), \kappa_n(f(\boldsymbol{\iota}(I)))) \\ &= m_{\mathcal{L},b}(\boldsymbol{\iota}^{-1}(f_{\lfloor P \rfloor_n}(\boldsymbol{\iota}(I))), \kappa_n(f_{\lfloor P \rfloor_n}(\boldsymbol{\iota}(I)))) \leq b^{-n} \end{aligned}$$

□

Figure 7.4 shows a Prolog implementation computing the set of constant pieces. To represent the hyper cube $u \in \mathfrak{P}^d_{b,n}(P)$, we use the coordinate of the corner closest to $\mathbf{0}$ and the side length. Because this length is $\frac{1}{b-1} \cdot b^{-n}$ for all hyper cubes of level n, it suffices to compute the set of coordinates. From Lemma 7.2.15 and Proposition 7.2.17, we conclude that this can by done by computing all interpretations over atoms up to level n and use their embedded values. This is possible because their embeddings coincide with the desired coordinates. A run of this is shown in Example 7.4.1. It shows the set of constant pieces of level 3.

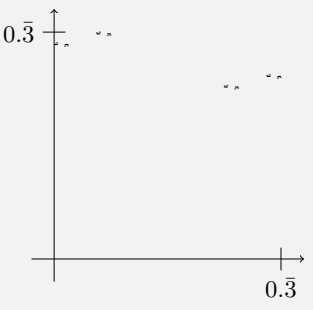
```
1 constantPieces(B, Program, LM, N, Positions, Length) :-
2     HB := { Atom | L in [1..N], level(LM, Atom, L:_) },
3     Length := 1/((B-1)*B^N),
4     Offset := Length / 2,
5     Positions := { X/Y | I := subset(HB),
6                          X := iota(LM, B, I),
7                          J := tp(Program, HB, I),
8                          Y := iota(LM, B, J) + Offset }.
```

Figure 7.4: *Implementation to compute a set of constant pieces which approximate a given T_P-operator up to level* `N`*: For a given base* `B`*, a program* `Program` *and some level* `N`*, a set of positions and a length is computed. Each piece* `X/Y` *consists of a hyper square with a side length of* `Length` *that is positioned at* `X`*.* `Y` *is unified with the value of $f_{\lfloor P \rfloor_n}$ for every dimension.* `LM` *denotes the level mapping as discussed in Section 7.2.*

From Proposition 7.4.4 we learn that it is sufficient to approximate $\mathfrak{P}^d_{b,n,P}$ up to ε_n. This is done by sigmoidal networks in the following section and afterwards using RBF and vector based networks.

Example 7.4.1 The plot of f_P for our running example as shown in Example 7.3.1 together with an approximation by constant pieces. On the left, the call and result of the Prolog implementation is shown, and on the right the resulting graph. This approximation is done for level 3.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   constantPieces(4, Prog, eo:1, 3, P, L).

P = [ 0.000 / 0.315,  0.016 / 0.315,
      0.062 / 0.331,  0.078 / 0.331,
      0.250 / 0.253,  0.266 / 0.253,
      0.312 / 0.268,  0.328 / 0.268 ]
L = 0.005
```



7.4.2 Approximation by Sigmoidal Networks

In this section we show how to approximate a given T_P-operator using a feed-forward network with sigmoidal activation functions in the hidden layer. For 1-dimensional embeddings, we construct a set of step functions such that their sum coincides with the function $\mathfrak{P}^1_{b,n,P}$ as defined above. Those step functions are then transformed into sigmoidals. Finally we show how to set up a network such that the network computes the sum of those sigmoidal functions.

In the one-dimensional case, we can sort the constant pieces introduced above from left to right, i.e., ascending with respect to their positions. A step function $\Theta^y_{0,x}$ can be placed between two neighbouring pieces, such that y is the difference between the heights of both pieces and x is the centre between them. I.e., we can compute $\mathfrak{P}^1_{b,n,P}$ by adding all those step functions and the height of the first piece.

Definition 7.4.5 *Let P be a covered logic program, $b > 2$, $n \geq 1$ and $\mathfrak{P}^1_{b,n}(P)$ be defined as above. Furthermore, we assume l to be the length of the u for $(u, v) \in \mathfrak{P}^1_{b,n}(P)$. Let $C = [p_1/v_1, p_2/v_2, \ldots, p_n/v_{2^n}]$ be the sorted sequence (ascending wrt. p_i) of left borders of the intervals u_i and let v_i be the corresponding output values, i.e., for each p_i/v_i there is $([p_i, p_i + l], v_i) \in \mathfrak{P}^1_{b,n}(P)$. Then we define the corresponding set of step functions as follows*

$$\mathfrak{T}_{b,n}(P) := \left\{ \Theta^h_{0,p} \mid p_i/v_i, p_{i+1}/v_{i+1} \in C \text{ and } h = v_i - v_{i+1} \text{ and } p = \frac{p_i + l + p_{i+1}}{2} \right\}$$

We define the corresponding approximation as follows:

$$\mathfrak{T}_{b,n,P} : \mathbb{R} \to \mathbb{R} : x \mapsto v_1 + \sum_{\Theta^h_{0,p} \in \mathfrak{T}_{b,n}(P)} \Theta^h_{0,p}(x)$$

Obviously, $\mathfrak{T}_{b,n,P}$ and $\mathfrak{P}^1_{b,n,P}$ coincide on $\mathfrak{C}^1_b$ as shown in the following lemma.

Lemma 7.4.6 *Let $\mathfrak{T}_{b,n,P}$ and $\mathfrak{P}^1_{b,n,P}$ be defined as above. Then we find for all $x \in \mathfrak{C}^1_b$:*

$$\mathfrak{T}_{b,n,P}(x) = \mathfrak{P}^1_{b,n,P}(x)$$

Proof As in Definition 7.4.5, we use $C = [p_1/v_1, p_2/v_2, \ldots, p_n/v_{2^n}]$ as the sorted sequence (ascending wrt. p_i) of starting points of the u_i and v_i be the corresponding

values. Furthermore, we assume that $x \in u_m$, i.e., m is the greatest index such that $p_m \leq x$. Then we find:

$$\mathfrak{T}_{b,n,P}(x) = v_1 + \sum_{\Theta^h_{0,p} \in \mathfrak{T}_{b,n}(P)} \Theta^h_{0,p}(x) = v_1 + \sum_{\substack{\Theta^h_{0,p} \in \mathfrak{T}_{b,n}(P) \\ p \leq x}} h$$

$$= v_1 + v_2 - v_1 + v_3 - v_2 + \ldots + v_m - v_{m-1} = v_m = \mathfrak{P}^1_{b,n,P}(x)$$

□

The following result is an immediate consequence of this equality and Proposition 7.4.4 from the previous section. It states that we can approximate $T_{\lfloor P \rfloor_n}$ using $\mathfrak{T}_{b,n,P}(x)$.

Corollary 7.4.7 *Let P be some covered logic program, $b > 2$, $n \geq 0$, let $\tilde{n}$, $T_{\lfloor P \rfloor_n}$ and $\mathfrak{T}_{b,\tilde{n},P}$ be as defined above. Then we find $m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(\mathfrak{T}_{b,\tilde{n},P}(\boldsymbol{\iota}(I)))) \leq b^{-n}$ for all $I \in \mathcal{I}_\mathcal{L}$.*

Figure 7.5 shows a Prolog implementation to compute the set of step functions and the additional offset. The set of constant pieces is ordered and the offset of the first piece is extracted. Finally, we compute the parameters for a step function for each neighbouring pair of constant pieces. Additionally, we store the distance between the step and the next constant piece. This distance is needed while converting the steps into sigmoidals below.

```
1 steps(B, Program, LM, N, Steps, Offset) :-
2     constantPieces(B, Program, LM, N, Positions, Length),
3     SPositions := sort(Positions),
4     SPositions = [0/Offset|_],
5     Steps := { P/H/D | append(_, [P1/V1,P2/V2|_], SPositions),
6                        P := (P1 + Length + P2) / 2,
7                        H := V2 - V1, \+ H =:= 0,
8                        D := P2 - P                              }.
```

Figure 7.5: *Implementation to compute a set of approximating step functions for a given T_P-operator up to level* `N`*: For a given base* `B`*, a program* `Program` *and some level* `N`*, a set of step functions and the offset is computed. Each step-descriptions* `P/H/D` *corresponds to the step function $\Theta^{\mathtt{H}}_{0,\mathtt{P}}$. The* `D` *contains the distance between* `P` *and the closest constant piece. As above, we use* `LM` *to refer to the level mapping.*

To construct an approximating network of sigmoidal units, we approximate the step functions just constructed by sigmoidal functions as introduced in Definition 2.5.7.

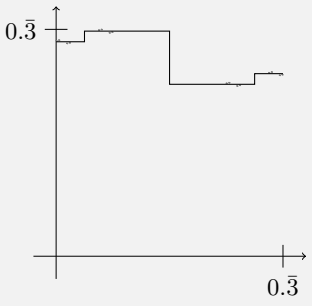
$$\mathrm{sigm}^h_{p,s}(x) := \frac{h}{1 + e^{-s(x-p)}}$$

Figure 7.6 shows a step functions and some corresponding sigmoidals. Obviously, there is some difference between a step function and a corresponding sigmoidal. But, similar to the ideas described in Chapter 3, we can scale the sigmoidal such that this error is below a certain value – at least on some intervals. For this we need the above mentioned distance d between a step and the next constant piece. To limit the approximation error by some ε_s, we need to make sure that the following condition is satisfied:

$$\left|\Theta^h_{0,p}(d) - \mathrm{sigm}^h_{p,s}(d)\right| \leq \varepsilon_s \tag{7.2}$$

Example 7.4.2 The plot of f_P for our running example as shown in Example 7.3.1 together with an approximation by step functions. On the left, the call and result of the Prolog implementation is shown. Each step-descriptions `X/Y/D` corresponds to the step function $\Theta^{\mathtt{Y}}_{0,\mathtt{X}}$. As mentioned above, the distance D is used only later and can be ignored here. On the right the resulting graph is depicted. As in Example 7.4.1, this approximation is done for level 3.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   steps(4, Prog, eo:1, 3, Steps, Offset).

Steps = [ 0.042 / 0.016 / 0.021,
          0.167 /-0.078 / 0.083,
          0.292 / 0.016 / 0.021]
Offset = 0.315
```



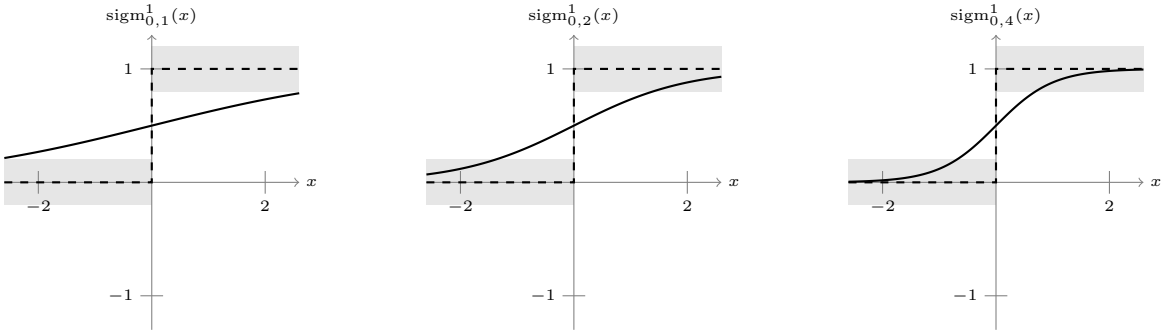


Figure 7.6: *Three sigmoidal approximations of the step function $\Theta^1_{0,0}$: The sigmoidals are shown for the scaling factors 1,2 and 4. The grey areas indicate an ε_s-tube of 0.2 around the step function.*

For simplicity and without loss of generality, we assume $d > 0$ and $p = 0$. The difference between the step and sigmoidal function can be computed as follows:

$$\left|\Theta^h_{0,0}(d) - \mathrm{sigm}^h_{0,s}(d)\right| = \left|h - \frac{h}{1+e^{-sd}}\right| = |h| - \frac{|h|}{1+e^{-sd}} \tag{7.3}$$

Using this equation, Condition (7.2) can be transformed into a condition on s as follows:

$$\begin{aligned}
|h| - \frac{|h|}{1+e^{-sd}} &\leq \varepsilon_s \\
|h| - \varepsilon_s &\leq \frac{|h|}{1+e^{-sd}} \\
1 + e^{-sd} &\leq \frac{|h|}{|h| - \varepsilon_s} \\
-sd &\leq \ln\left(\frac{|h|}{|h| - \varepsilon_s} - 1\right) \\
s &\geq -\frac{\ln\left(\frac{|h|}{|h|-\varepsilon_s} - 1\right)}{d}
\end{aligned}$$

The condition $h > \varepsilon_s$ is not severe, because we can simply ignore all steps with $h \leq \varepsilon_s$. We define a set of sigmoidal functions $\mathfrak{S}_{b,n}(P)$ such that their sum $\mathfrak{S}_{b,n,P} : \mathbb{R} \to \mathbb{R}$ approximates the function $\mathfrak{T}_{b,n,P}$ up to ε_n. For this we divide ε by the number t of steps with non-zero height. Each of those step functions $\Theta^h_{0,p}$ is approximated by a sigmoidal $\mathrm{sigm}^h_{p,s}$ such that Condition (7.2) is satisfied. Please note, that we could also remove all steps with height $h < \varepsilon_s$ but those are kept for simplicity of the implementation and the proofs.

Definition 7.4.8 *Let P be a covered logic program, let $b > 2$ and let $n \geq 1$. Let $\mathfrak{T}_{b,n}(P)$ and v_1 be as in Definition 7.4.5 and let $t = |\{\Theta^h_{0,p} \mid \Theta^h_{0,p} \in \mathfrak{T}_{b,n}(P) \text{ and } h \neq 0\}|$, i.e., the number of non-zero steps. Furthermore, let $d(x)$ denote the distance from x to the next constant piece and let $\varepsilon_s = \frac{\varepsilon_n}{t}$ for ε_n as above. Then we define the corresponding set of sigmoidal functions as follows*

$$\mathfrak{S}_{b,n}(P) := \left\{ \mathrm{sigm}^h_{p,s} \mid \Theta^h_{0,p} \in \mathfrak{T}_{b,n}(P) \text{ with } |h| > \varepsilon_s \text{ and } s := -\frac{\ln\left(\frac{|h|}{|h|-\varepsilon_s} - 1\right)}{d(p)} \right\}$$

We define the corresponding approximating as follows:

$$\mathfrak{S}_{b,n,P} : \mathbb{R} \to \mathbb{R} : x \mapsto v_1 + \sum_{\mathrm{sigm}^h_{p,s} \in \mathfrak{S}_{b,n}(P)} \mathrm{sigm}^h_{p,s}(x)$$

Lemma 7.4.9 *Let $\mathfrak{S}_{b,n,P}$, $\mathfrak{P}^1_{b,n,P}$, t, ε and $d(x)$ be defined as above. Then we find:*

$$\left|m_{\mathsf{m}}(\mathfrak{P}^1_{b,n,P}(x), \mathfrak{S}_{b,n,P}(x))\right| \leq \varepsilon \quad \text{for all } x \in \mathfrak{C}^1_b.$$

Proof Using Lemma 7.4.6 we know $\mathfrak{T}_{b,n,P}(x) = \mathfrak{P}^1_{b,n,P}(x)$, and hence, it suffices to show

$$\left|m_{\mathsf{m}}(\mathfrak{T}_{b,n,P}(x), \mathfrak{S}_{b,n,P}(x))\right| \leq \varepsilon$$

Because for every step function exists either a approximating sigmoidal or the height was smaller than ε_s, we can rewrite this to:

$$\sum_{\mathrm{sigm}^h_{p,s}(x)\in\mathfrak{S}_{b,n}(P)} m_{\mathsf{m}}(\Theta^h_{0,p}(x), \mathrm{sigm}^h_{p,s}(x)) \leq \varepsilon$$

Using Equation (7.3), we can simplify this as follows:

$$\begin{aligned}
\sum_{\mathrm{sigm}^h_{p,s}(x)\in\mathfrak{S}_{b,n}(P)} m_{\mathsf{m}}(\Theta^h_{0,p}, \mathrm{sigm}^h_{p,s}(x)) &= \sum_{\mathrm{sigm}^h_{p,s}(x)\in\mathfrak{S}_{b,n}(P)} \left|\Theta^h_{0,0}(d(p)) - \mathrm{sigm}^h_{0,s}(d(p))\right| \\
&= \sum_{\mathrm{sigm}^h_{p,s}(x)\in\mathfrak{S}_{b,n}(P)} |h| - \frac{|h|}{1+e^{-sd(p)}} \\
&\leq \sum_{\mathrm{sigm}^h_{p,s}(x)\in\mathfrak{S}_{b,n}(P)} |h| - \frac{|h|}{1+e^{\frac{\ln\left(\frac{|h|}{|h|-\varepsilon_s}-1\right)}{d(p)}\cdot d(p)}} \\
&= \sum_{\mathrm{sigm}^h_{p,s}(x)\in\mathfrak{S}_{b,n}(P)} |h| - \frac{|h|}{\frac{|h|}{|h|-\varepsilon_s}} \\
&= \sum_{\mathrm{sigm}^h_{p,s}(x)\in\mathfrak{S}_{b,n}(P)} \varepsilon_s \\
&\leq t\cdot\varepsilon_s = \varepsilon
\end{aligned}$$

□

As above, we can now conclude that $\mathfrak{S}_{b,n,P}$ can indeed be used to compute $T_{\lfloor P\rfloor_n}$. Using κ_n from above, we can map the output of $\mathfrak{S}_{b,n,P}$ back to interpretations, and find that $T_{\lfloor P\rfloor_n}(I)$ and $\kappa_n(\mathfrak{S}_{b,\tilde{n},P}(\boldsymbol{\iota}(I)))$ agree on all atoms up to level n.

Corollary 7.4.10 *Let P be some covered logic program, $n \geq 0$, let $\tilde{n}$, $T_{\lfloor P\rfloor_n}$ and $\mathfrak{S}_{b,\tilde{n},P}$ be as defined above. Then we find*

$$m_{\mathcal{L},b}(T_{\lfloor P\rfloor_n}(I), \kappa_n(\mathfrak{S}_{b,\tilde{n},P}(\boldsymbol{\iota}(I)))) \leq b^{-n} \quad \text{for all } I \in \mathcal{I}_{\mathcal{L}}.$$

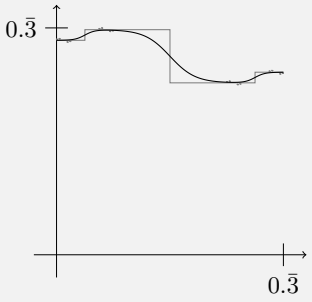
Figure 7.7 shows the Prolog implementation to compute the set of sigmoidal functions. It is based on the `steps/6`-predicate shown in Figure 7.5. For each step, an approximating sigmoidal function is constructed.

```
1 sigmoidals(B, Program, LM, N, [], Offset) :-
2     steps(B, Program, LM, N, [], Offset), !.
3
4 sigmoidals(B, Program, LM, N, Sigmoidals, Offset) :-
5     steps(B, Program, LM, N, Steps, Offset),
6     Epsilon := 1/(2*B^N*(B-1)),
7     EpsilonS := Epsilon / size(Steps),
8     Sigmoidals := { P/H/S | P/H/D in Steps, abs(H) > EpsilonS,
9                             S := -log((abs(H)/(abs(H)-EpsilonS))-1)/abs(D) }.
```

Figure 7.7: *Implementation to compute a set of approximating sigmoidal functions for a given T_P-operator up to level* `N`*: First, the set of step functions is computed using* `steps/6`*-predicate as shown in Figure 7.5. Afterwards, the set of sigmoidals is constructed following Definition 7.4.8.*

Example 7.4.3 A sigmoidal approximation of f_P for our running example. On the left, the call and result of the Prolog implementation is shown. Each triple `P/H/S` corresponds to the sigmoidal function $\mathrm{sigm}^{\mathtt{H}}_{\mathtt{P},\mathtt{S}}$. On the right the resulting graph is depicted. It contains the step functions from Example 7.4.2 overlayed with the resulting sigmoidal approximation. As above, the approximation is done for level 3.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   sigmoidals(4, Prog, eo:1, 3, S, O).

S = [ 0.042 / 0.016 / 135.994,
      0.167 /-0.078 /  53.864,
      0.292 / 0.016 / 135.994]
O = 0.315
```



Now we can construct an approximating network. Each sigmoidal function in $\mathfrak{S}_{b,n}(P)$ is represented by a hidden unit. The weights are set up such that they parametrise the sigmoidals appropriately. The input unit is used to feed some (embedded) interpretation into the network and the output unit computes the weighted sum.

Definition 7.4.11 *Let P be a covered logic program, $b > 2$, $n \geq 1$ and let $\mathfrak{S}_{b,n}(P)$ and v_1 be as in Definition 7.4.8. Then we define the corresponding sigmoidal network $\mathfrak{S}_{b,n,P}$ as follows:*

$$
\begin{aligned}
\mathfrak{S}_{b,n,P} &:= \langle \mathcal{U}_{\mathsf{inp}}, \mathsf{id}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \mathrm{sigm}^1_{0,1}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}}, \mathsf{id} \rangle \quad \textit{with} \\
\mathcal{U}_{\mathsf{inp}} &:= \{\mathsf{inp}(0)\} \\
\mathcal{U}_{\mathsf{hid}} &:= \{\mathsf{hid}_i(\theta) \mid \mathrm{sigm}^h_{p,s} \in_i \mathfrak{S}_{b,n}(P) \textit{ and } \theta = -p \cdot s\} \\
\mathcal{U}_{\mathsf{out}} &:= \{\mathsf{out}(v_1)\} \\
\omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}} &: \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \to \mathbb{R} : (\mathsf{inp}, \mathsf{hid}_i) \mapsto s \textit{ with } \mathrm{sigm}^h_{p,s} \in_i \mathfrak{S}_{b,n}(P) \\
\omega_{\mathsf{hid}\blacktriangleright\mathsf{out}} &: \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}} \to \mathbb{R} : (\mathsf{hid}_i, \mathsf{out}) \mapsto h \textit{ with } \mathrm{sigm}^h_{p,s} \in_i \mathfrak{S}_{b,n}(P)
\end{aligned}
$$

The following lemma links the sigmoidal approximation and the network function of the corresponding $\mathfrak{S}_{b,n,P}$.

Lemma 7.4.12 *Let $f_{\mathfrak{S}}$ be the network function of $\mathfrak{S}_{b,n,P}$ as introduced in Definition 2.5.15. Then we find $f_{\mathfrak{S}}(x) = \mathfrak{S}_{b,n,P}(x)$ for all $x \in \mathfrak{U}^1_b$.*

Proof This follows from the fact that each sigmoidal function $\mathrm{sigm}^{h_i}_{p_i,s_i} \in \mathfrak{S}_{b,n}(P)$ is represented by one hidden unit and the weights are set up such that its activation coincides with $\mathrm{sigm}^{h_i}_{p_i,s_i}(x)$ if the input unit is initialised to x. Because the output unit computes the weighted sum, we find $f_{\mathfrak{S}}(x) = \mathfrak{S}_{b,n,P}(x)$ for all $x \in \mathfrak{U}^1_b$. □

A Prolog implementation constructing a sigmoidal network is shown in Figure 7.8. First the set of approximating sigmoidal functions is computed and afterwards, it is turned into an equivalent network following Definition 7.4.11.

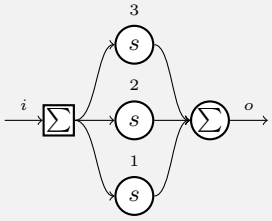
We are now in a position to restate the following result. Using our definitions from above and Lemma 7.4.12 we can conclude that there is a sigmoidal network that can be used to approximate the T_P-operator for covered logic programs up to an arbitrary level.

```
sigmoidalNetwork(B, Program, LM, N, Network) :-
    sigmoidals(B, Program, LM, N, Sigmoidals, Offset),

    Ui := { unit(i, 0) },
    Uh := { unit(I, T) | P/_H/S in Sigmoidals at I, T := -1*P*S },
    Uo := { unit(o, Offset) },

    I2H := { i/I/S | _/_/S in Sigmoidals at I },
    H2O := { I/o/H | _/H/_ in Sigmoidals at I },

    Network = ffn3(Ui, id, I2H, Uh, sigm, H2O, Uo, id).
```

Figure 7.8: *Implementation to construct an approximating sigmoidal network for a given T_P-operator up to level* `N`*: First, the set of sigmoidal functions is computed using* `sigmoidals/6`*-predicate as shown in Figure 7.7. Afterwards, this set is turned into an equivalent network according to Definition 7.4.11.*

Example 7.4.4 A sigmoidal network approximating f_P for our running example. On the left, the call and result of the Prolog implementation is shown. On the right the resulting network is depicted. The hidden layer contains 3 units computing the sigmoidal function. And the output layer unit computes the identity, i.e., it outputs the weighted sum of the hidden units outputs. As above, the approximation is done for level 3.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   sigmoidalNetwork(4, Prog, eo:1, 3, N).

N = ffn3([unit(i,0.00)], id,
     [i/1/135.994, i/2/53.864, i/3/135.994],
     [unit(1,-5.67), unit(2,-8.98), unit(3,-39.66)], sigm,
     [1/o/0.016, 2/o/-0.078, 3/o/0.016],
     [unit(o,0.32)], id)
```



Theorem 7.4.13 *Let P be a covered logic program with consequence operator T_P. Let $b > 2$ and $n \geq 0$ and let $\tilde{n}$ be as defined above. Then there exists a three layer feed-forward sigmoidal network $\mathfrak{S}_{b,\tilde{n},P}$ such that its network function $f_{\mathfrak{S}}$ approximates $T_{\lfloor P \rfloor_n}$, i.e., we find*

$$m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(f_{\mathfrak{S}}(\iota(I)))) \leq b^{-n} \quad \text{for all } I \in \mathcal{I}_{\mathcal{L}}.$$

Proof This theorem follows from Lemma 7.4.12 and Corollary 7.4.10 above. □

This last theorem, is essentially the same as shown in [HKS99], but our version here is constructive. Therefore, we finally close the gap between logic consequence operators and functions computed in a connectionist system.

In the following two sections we discuss other approaches to do this. Namely by means of radial basis function networks and vector-based networks. In Section 7.5, we investigate the iteration of the approximation. I.e., there we study what happens if we feed back the output of the network into the input layer repeatedly. Our sigmoidal network can not only be used to approximate one application of T_P, but also to approximate the least fixed point, provided the consequence operator is contractive.

7.4.3 Approximation by Radial Basis Function Networks

Now, we focus on the construction of radial basis function networks as introduced in Definition 2.5.20. In particular, we show how to approximate the 1-dimensional embedded T_P operator using networks with raised cosine activation function. As introduced in Definition 2.5.12, we define the raised cosine function as follows:

$$\operatorname{rcos}^h_{p,w} : \mathbb{R} \to \mathbb{R} : x \mapsto \begin{cases} \frac{h}{2} \cdot \left(1 + \cos\left(\pi \cdot \frac{x-p}{w}\right)\right) & \text{if } |x-p| \leq w \\ 0 & \text{otherwise} \end{cases}$$

Please note, that if two raised cosines $\operatorname{rcos}^h_{p_1,w}$ and $\operatorname{rcos}^h_{p_2,w}$ with $|p_1 - p_2| = w$ are added, we obtain a function that is constant on the interval $[p_1, p_2]$. Therefore, we can represent each constant piece from Definition 7.4.3 by two raised cosines.

Definition 7.4.14 *Let P be a covered logic program, $b > 2$, $n \geq 1$ and $\mathfrak{P}^1_{b,n}(P)$ be defined as above. Then we define the corresponding set of raised cosines as follows*

$$\mathfrak{R}_{b,n}(P) := \left\{ \operatorname{rcos}^v_{p_1,l}, \operatorname{rcos}^v_{p_2,l} \,|([p_1,p_2], v) \in \mathfrak{P}^1_{b,n}(P) \text{ and } l = p_2 - p_1 \right\}$$

We define the corresponding approximating as follows:

$$\mathfrak{R}_{b,n,P} : \mathbb{R} \to \mathbb{R} : x \mapsto \sum_{\operatorname{rcos}^h_{p,w} \in \mathfrak{R}_{b,n}(P)} \operatorname{rcos}^h_{p,w}(x)$$

After showing that $\mathfrak{R}_{b,n,P}(x) = \mathfrak{P}^1_{b,n,P}(x)$ holds for all $x \in \mathfrak{C}^1_b$, we obtain immediately the desired result that we can approximate T_P up to some level n using raised cosines as stated in Corollary 7.4.16 below.

Proposition 7.4.15 *Let P be a covered logic program, $b > 2$, $n \geq 1$, let $\mathfrak{P}^1_{b,n,P}$ be as in Definition 7.4.3 and let $\mathfrak{R}_{b,n,P}$ be defined as above. Then we find*

$$\mathfrak{R}_{b,n,P}(x) = \mathfrak{P}^1_{b,n,P}(x) \text{ for all } x \in \mathfrak{C}^1_b.$$

Proof By construction, we find that for each constant piece $[p_1, p_2] \in \mathfrak{P}^1_{b,n}(P)$, there are two raised cosines with centres p_1 and p_2. To show the equivalence of $\mathfrak{R}_{b,n,P}$ and $\mathfrak{P}^1_{b,n,P}$, we need to show that (a) those two raised cosines add up to a constant function on the interval $[p_1, p_2]$ and (b) that they do not interfere with other intervals. (a) follows by definition, because for $x \in [p_1, p_2]$ and $l = p_2 - p_1$ we find:

$$\begin{aligned}
\mathrm{rcos}^v_{p_1,l}(x) + \mathrm{rcos}^v_{p_1,l}(x) &= \frac{v}{2} \cdot \left(1 + \cos\left(\pi \cdot \frac{x - p_1}{l}\right)\right) + \frac{v}{2} \cdot \left(1 + \cos\left(\pi \cdot \frac{x - p_2}{l}\right)\right) \\
&= \frac{v}{2} \cdot \left(2 + \cos\left(\pi \cdot \frac{x - p_1}{l}\right) + \cos\left(\pi \cdot \frac{x - p_1 - l}{l}\right)\right) \\
&= \frac{v}{2} \cdot \left(2 + \cos\left(\pi \cdot \frac{x - p_1}{l}\right) - \cos\left(\pi \cdot \frac{x - p_1}{l}\right)\right) \\
&= \frac{v}{2} \cdot 2 = v
\end{aligned}$$

(b) follows from the facts that $\mathrm{rcos}^h_{p,l}(x) = 0$ for $|x - p| > l$ and that the distance between two neighbouring pieces is at least as big as l. □

Corollary 7.4.16 *Let P be some covered logic program, $n \geq 0$, let $\tilde{n}$, $T_{\lfloor P \rfloor_n}$ and $\mathfrak{R}_{b,\tilde{n},P}$ be as defined above. Then we find*

$$m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(\mathfrak{R}_{b,\tilde{n},P}(\boldsymbol{\iota}(I)))) \leq b^{-n} \quad \text{for all } I \in \mathcal{I}_{\mathcal{L}}.$$

Figure 7.9 shows the Prolog implementation to compute the set of raised cosines. It is based on the `constantPieces/6`-predicate shown in Figure 7.4. For each piece, two raised cosines are constructed according to Definition 7.4.14. A run together with the resulting plot is shown in Example 7.4.5.

```
1 rcos(B, Program, LM, N, RCos) :-
2     constantPieces(B, Program, LM, N, Positions, Length),
3     SPositions := sort(Positions),
4     RCos := { P/H/Length | P/H in SPositions } union
5             { P/H/Length | R/H in SPositions, P := R + Length }.
```

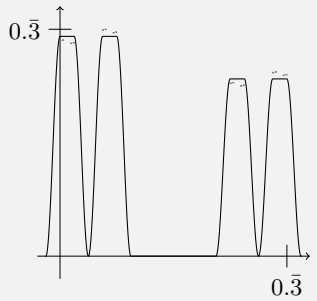
Figure 7.9: *Implementation to compute the set of approximating raised cosine functions for a given T_P-operator up to level* `N`*: First, the set of constant pieces is computed using* `constantPieces/6`*-predicate as shown in Figure 7.4. Afterwards, the set of raised cosines is constructed following Definition 7.4.14.*

We can now define a radial basis function network, based on this approximation. Each raised cosine function is represented by one unit in the hidden layer and the weights are set up such that the network function coincides with the approximation.

Definition 7.4.17 *Let P be a covered logic program, $b > 2$, $n \geq 1$ and let $\mathfrak{R}_{b,n}(P)$ be as in Definition 7.4.14. Then we define the corresponding radial basis function network $\mathfrak{R}_{b,n,P}$ as*

Example 7.4.5 An approximation of f_P for our running example using raised cosines. On the left, the call and result of the Prolog implementation is shown. Each triple `P/H/W` corresponds to the raised cosine $\mathrm{rcos}^{\mathtt{H}}_{\mathtt{P},\mathtt{W}}$. On the right the resulting graph is depicted. It contains the constant pieces from Example 7.4.1 overlayed with the resulting approximation. As above, the approximation is done for level 3.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   rcos(4, Prog, eo:1, 2, RC).

RC = [ 0.000 / 0.323 / 0.021,
       0.062 / 0.323 / 0.021,
       0.250 / 0.260 / 0.021,
       0.312 / 0.260 / 0.021,
       0.021 / 0.323 / 0.021,
       0.083 / 0.323 / 0.021,
       0.271 / 0.260 / 0.021,
       0.333 / 0.260 / 0.021 ]
```



follows:

$$
\begin{aligned}
\mathfrak{R}_{b,n,P} &:= \left\langle \mathrm{rcos}^{1}_{0,1}, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}} \right\rangle \quad \textit{with} \\
\mathcal{U}_{\mathsf{inp}} &:= \{\mathsf{inp}(0)\} \\
\mathcal{U}_{\mathsf{hid}} &:= \{\mathsf{hid}_i(w) \mid \mathrm{rcos}^{h}_{p,w} \in_i \mathfrak{R}_{b,n}(P)\} \\
\mathcal{U}_{\mathsf{out}} &:= \{\mathsf{out}(0)\} \\
\omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}} &: \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \to \mathbb{R} : (\mathsf{inp}, \mathsf{hid}_i) \mapsto p \ \textit{with} \ \mathrm{rcos}^{h}_{p,w} \in_i \mathfrak{R}_{b,n}(P) \\
\omega_{\mathsf{hid}\blacktriangleright\mathsf{out}} &: \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}} \to \mathbb{R} : (\mathsf{hid}_i, \mathsf{out}) \mapsto h \ \textit{with} \ \mathrm{rcos}^{h}_{p,w} \in_i \mathfrak{R}_{b,n}(P)
\end{aligned}
$$

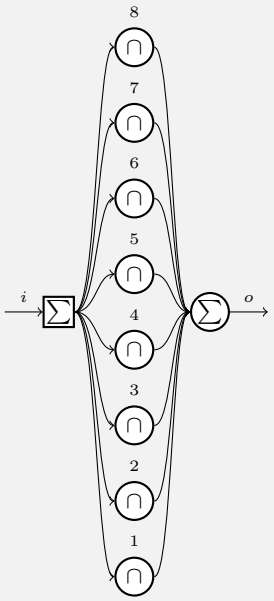
The Prolog code to construct an approximating network of raised cosines is shown in Figure 7.10. First the set of approximating raised cosines is computed. Afterwards, it is turned into an equivalent network following Definition 7.4.17. The result is shown in Example 7.4.6.

```
rcosNetwork(B, Program, LM, N, Network) :-
    rcos(B, Program, LM, N, RCos),

    Ui := { unit(i, 0) },
    Uh := { unit(I, Width) | P/H/Width in RCos at I },
    Uo := { unit(o, 0) },

    I2H := { i/I/P | P/_/_ in RCos at I },
    H2O := { I/o/H | _/H/_ in RCos at I },

    Network = rbf(rcos, Ui, I2H, Uh, H2O, Uo).
```

Figure 7.10: *Implementation to construct an approximating raised cosine network for a given T_P-operator up to level* `N`*: First, the set of raised cosines is computed using* `rcos/5`*-predicate as shown in Figure 7.9. Afterwards, this set is turned into an equivalent network according to Definition 7.4.17.*

Example 7.4.6 An RBF network of raised cosines approximating f_P for our running example. On the left, the call and result of the Prolog implementation is shown. On the right the resulting network is depicted. The hidden layer contains eight units computing the raised cosine. And the output layer unit computes the weighted sum of the hidden units outputs. The approximation is done for level 2.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   rcosNetwork(4, Prog, eo:1, 2, N).

N = rbf(rcos,
    [unit(i,0.00)],
    [i/1/0.000, i/2/0.062, i/3/0.250,
     i/4/0.312, i/5/0.021, i/6/0.083,
     i/7/0.271, i/8/0.333],
    [unit(1,0.02), unit(2,0.02), unit(3,0.02),
     unit(4,0.02), unit(5,0.02), unit(6,0.02),
     unit(7,0.02), unit(8,0.02)],
    [1/o/0.323, 2/o/0.323, 3/o/0.260,
     4/o/0.260, 5/o/0.323, 6/o/0.323,
     7/o/0.260, 8/o/0.260],
    [unit(o,0.00)])
```



The following lemma links the approximation by raised cosines and the network function of the corresponding $\mathfrak{R}_{b,n,P}$ in the desired way, by stating that both coincide.

Lemma 7.4.18 *Let $f_{\mathfrak{R}}$ be the input-output function of the $\mathfrak{R}_{b,n,P}$ introduced above. Then we find $f_{\mathfrak{R}}(x) = \mathfrak{R}_{b,n,P}(x)$ for all $x \in \mathfrak{U}_b^1$.*

Proof This follows from the fact that there is a unit for each raised cosine function $\mathrm{rcos}_{p,w}^h$ in $\mathfrak{R}_{b,n}(P)$. This unit does output $\mathrm{rcos}_{p,w}^1(x)$ if the input unit is initialised to output x. Because the output unit computes the weighted sum over those outputs multiplied with the corresponding h, we find $f_{\mathfrak{R}}(x) = \mathfrak{R}_{b,n,P}(x)$ for all $x \in \mathfrak{U}_b^1$. □

As for the sigmoidal networks above, we can now state the following theorem:

Theorem 7.4.19 *Let P be a covered logic program with consequence operator T_P. Let $b > 2$ and $n \geq 0$ and let $\tilde{n}$ be as defined above. Then there exists a radial basis function network $\mathfrak{R}_{b,\tilde{n},P}$ of raised cosines such that its network function $f_{\mathfrak{R}}$ can be used to compute $T_{\lfloor P \rfloor_n}$, i.e., we find*

$$m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(f_{\mathfrak{R}}(\boldsymbol{\iota}(I)))) \leq b^{-n} \quad \text{for all } I \in \mathcal{I}_{\mathcal{L}}.$$

Proof This follows from Lemma 7.4.18 and Corollary 7.4.16 from above. □

Theorem 7.4.19 tells us that there are radial basis function networks that approximate a given T_P-operator up to any given level of accuracy. Furthermore, we are able to construct such a network using the definitions and implementations presented above. But, those networks

have the following drawback: the approximation is not very smooth. I.e., small changes in the input may lead to big changes in the output as shown in Example 7.4.5. Unfortunately, this detains us from using those networks for an iteration of the approximation, because we cannot guarantee its convergence. Nevertheless, we can use those networks to approximate a single application of the T_P-operator. In the following section, we investigate a third possible approximation using vector-based networks.

7.4.4 Approximation by Vector-Based Networks

In this section, we discuss a third connectionist approach to approximate the T_P-operator of some given program. Here, we use vector-based networks as introduced in Definition 2.5.22. Again it is based on the piecewise constant approximation discussed in Section 7.4.1, but now for arbitrary dimensions. Using the locality of vector-based networks, we can represent each constant piece by a unit of the network. This can be done by setting the reference vectors to the centres of the constant pieces. Before defining a vector-based approximation, we briefly recall the main concepts.

Each hidden unit in a vector-based network is connected to all input units and to all output units. The weights from the input units encode the position of a so-called reference vector and the weights to the output units encode the output with respect to this unit. The hidden units compute the distance between their positions and the current input and the unit which is closest is chosen as the *winner*. The winner unit outputs 1, while all the other units output 0. This activation is further propagated to the output units, which compute the weighted sum. Alternatively, we can understand such a network as a set of reference vectors to which an output vector is attached. The output of the network is the output of the closest reference vector. We do not yet fix a distance measure in the following definition, but below we discuss different approaches.

Definition 7.4.20 *Let P be a covered logic program, $b > 2$, $n, d \geq 1$ and $\mathfrak{P}^d_{b,n}(P)$ be as in Definition 7.4.3, and let $m : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be a metric on $\mathbb{R}^d$. Then we define the corresponding set of reference vectors as follows*

$$\mathfrak{V}^{d,m}_{b,n}(P) := \left\{ (p, v) \mid (u, v) \in \mathfrak{P}^d_{b,n}(P) \text{ with } p = \boldsymbol{\iota}(I_u) + \left[\frac{b^{-n}}{b-1} \cdot \frac{1}{2} \right]^d \right\}$$

We define the corresponding approximating as follows:

$$\mathfrak{V}^{d,m}_{b,n,P} : \mathbb{R} \to \mathbb{R} : x \mapsto v \text{ with } (p, v) := \operatorname*{argmin}_{(p,v) \in \mathfrak{V}^{d,m}_{b,n}(P)} m(x, p)$$

Please note that the function $\mathfrak{V}^{d,m}_{b,n,P}$ is not necessarily well defined, because there could be several closest reference vectors (p_1, v_1) and (p_2, v_2) for some $x \in \mathbb{R}$ with different output values $v_1 \neq v_2$. This can be solved by assuming the set $\mathfrak{V}^{d,m}_{b,n}(P)$ to be ordered and choosing the first reference vector.

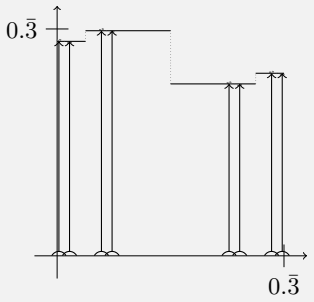
A Prolog implementation to compute the set of reference vectors is shown in Figure 7.11. As in the previous sections, it is based on the `constantPieces/6`-predicate shown in Figure 7.4. For each piece, a reference vector is added according to Definition 7.4.20. It is located in the centre of the hyper cube u and the output is set to the corresponding output v. A run together with the resulting plot is shown in Example 7.4.7.

```
vb(B, Program, LM, N, RVs) :-
    constantPieces(B, Program, LM, N,  Positions,  Length),
    RVs := { P/V | U/V in Positions, P := U + Length/2 }.
```

Figure 7.11: *Implementation to compute the set of approximating reference vectors for a given T_P-operator up to level* `N`*: First, the set of constant pieces is computed using* `constantPieces/6`*-predicate as shown in Figure 7.4. Afterwards, the set of reference vectors is constructed following Definition 7.4.20.*

Example 7.4.7 A vector-bases approximation of f_P for our running example. On the left, the call and result of the Prolog implementation is shown. Each triple `P/V` corresponds to a reference vector `(P,V)` at position `P` and with output `V`. On the right the resulting graph is depicted. It contains the graph overlaid with the resulting approximation. Each reference vector (p, v) is depicted as an arrow at position p pointing to v. As above, the approximation is done for level 3.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   vb(4, Prog, eo:1, 3, RVs).

RVs = [ 0.003 / 0.315,  0.018 / 0.315,
        0.065 / 0.331,  0.081 / 0.331,
        0.253 / 0.253,  0.268 / 0.253,
        0.315 / 0.268,  0.331 / 0.268 ]
```



Before constructing a connectionist system, we discuss the distance measure m to be used in this approximation. As a first approach, we can use the usual d-dimensional Euclidean norm from Definition 2.2.2:

$$m_{\mathfrak{E}} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} : ((x_1, \ldots, x_d), (y_1, \ldots, y_d)) \mapsto \sqrt{\sum_{i=1}^{d} (x_i - y_d)^2} \tag{7.4}$$

Proposition 7.4.21 *Let P be a covered logic program, $b > 2$, $n, d \geq 1$, let $\mathfrak{P}^d_{b,n,P}$ be as in Definition 7.4.3 and let $\mathfrak{V}^{d,m_{\mathfrak{E}}}_{b,n,P}$ be defined as above for the Euclidean metric $m_{\mathfrak{E}}$. Then we find*

$$\mathfrak{V}^{d,m_{\mathfrak{E}}}_{b,n,P}(x) = \mathfrak{P}^d_{b,n,P}(x) \text{ for all } x \in \mathfrak{C}^d_b.$$

Proof (sketch) Looking at the definition of $\mathfrak{V}^{d,m_{\mathfrak{E}}}_{b,n}(P)$, we find that there is a reference vector (p_u, v_u) located in the centre (wrt. the Euclidean norm) of every u for $(u, v_u) \in \mathfrak{P}^d_{b,n}(P)$. For all $(u, v_u) \in \mathfrak{P}^d_{b,n}(P)$ and all $x \in u$, we find that the closest reference vector is (p_u, v_u) and hence $\mathfrak{V}^{d,m_{\mathfrak{E}}}_{b,n,P}(x) = \mathfrak{P}^d_{b,n,P}(x)$ for all $x \in \mathfrak{C}^d_b$. □

Alternatively we could use the maximum metric on $\mathbb{R}^d$ introduced in Definition 2.2.3:

$$m_{\mathfrak{m}} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} : ((x_1, \ldots, x_d), (y_1, \ldots, y_d)) \mapsto \max_{1 \leq i \leq d} \sqrt{(x_i - y_d)^2} \tag{7.5}$$

Proposition 7.4.22 *Let P be a covered logic program, $b > 2$, $n, d \geq 1$, let $\mathfrak{P}^d_{b,n,P}$ be as in Definition 7.4.3 and let $\mathfrak{V}^{d,m_\mathsf{m}}_{b,n,P}$ be defined as above for the maximum metric m_m. Then we find*

$$\mathfrak{V}^{d,m_\mathsf{m}}_{b,n,P}(x) = \mathfrak{P}^d_{b,n,P}(x) \text{ for all } x \in \mathfrak{C}^d_b.$$

Proof (sketch) This can be shown analogously to Proposition 7.4.21, because the reference vectors position are also the centres wrt. the maximum metric. □

In [Wit06, BHHW07], a third approach was presented, which employs some features of the domain $\mathfrak{C}^d_b$. It is based on the following ideas:

- If a reference vector is the only vector in a certain hyper-cube then it should be considered as default vector.
- Among those vectors which are located in the same hyper-cube the one closest with respect to the maximum norm should be the winner.

Those ideas are formalised as $m_\mathfrak{U}$ in Definition 7.4.23. Please note, that the formalisation presented below differs from those presented in [Wit06, BHHW07], but should be equivalent in the sense that the same reference vectors would be winners. The previous formalisation did not define metrics, therefore a new version is presented here. In Proposition 7.4.24, we show that $m_\mathfrak{U}$ as defined below is a metric.

Definition 7.4.23 *For $b > 2$ and $d \geq 1$, we define the mapping $m_\mathfrak{U} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ as follows:*

$$\begin{aligned}
m_\mathfrak{U} &: \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \\
&\quad (x,y) \mapsto \begin{cases} 0 & \text{if } x = y \\ g(x,y) + m_\mathsf{m}(x,y) & \text{otherwise} \end{cases} \quad \text{with} \\
g &: \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{N} \\
&\quad (x,y) \mapsto \begin{cases} 3 & \text{if } x \notin \mathfrak{U}^d_b \text{ or } y \notin \mathfrak{U}^d_b \\ 2 & \text{if for } i > l(x,y) \text{ there are } u_x, u_y \in \Omega^i\left(\mathfrak{U}^d_b\right) \text{ with } x \in u_x \text{ and } y \in u_y \\ 1 & \text{if for } i > l(x,y) \text{ there is } u \in \Omega^i\left(\mathfrak{U}^d_b\right) \text{ with } x \in u \text{ or } y \in u \\ 0 & \text{otherwise} \end{cases} \\
l &: \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{N} \\
&\quad (x,y) \mapsto \max_{i \in \mathbb{N}} \left\{ i \mid u \in \Omega^i\left(\mathfrak{U}^d_b\right) \text{ with } x \in u \text{ and } y \in u \right\}
\end{aligned}$$

Proposition 7.4.24 *$m_\mathfrak{U}$ as introduced in Definition 7.4.23 is a metric.*

Proof To show that $m_\mathfrak{U}$ is a metric we need to show that it satisfies the three conditions stated in Definition 2.2.1, i.e., for all $x, y, z \in \mathbb{R}^d$ we must find: (i) $m_\mathfrak{U}(x,x) = 0$, (ii) $m_\mathfrak{U}(x,y) = m_\mathfrak{U}(y,x)$, and (iii) $m_\mathfrak{U}(x,z) \leq m_\mathfrak{U}(x,y) + m_\mathfrak{U}(y,z)$. Condition (i) and (ii) follow immediately from the definition of $m_\mathfrak{U}$.To show (iii), we will show that it holds for g provided $x \neq y$. Using the fact that $m_\mathsf{m}(x,y)$ is bounded by $\frac{1}{b-1} < 1$, we can conclude that condition (iii) holds for $m_\mathfrak{U}$ as well. The following table shows possible values for $g(x,z)$ provided the given values for $g(x,y)$ and $g(y,z)$:

$g(y,z)$ \ $g(x,y)$	0	1	2	3
0	0	1	2	3
1	1	0..2	0..2	3
2	2	0..2	0..2	3
3	3	3	3	3

Obviously, we find $g(x,z) \leq g(x,y) + g(y,z)$ for all combinations. Therefore, we can conclude that $m_{\mathfrak{U}}$ as introduced above is a metric. □

Lemma 7.4.25 *Let P be a covered logic program, $b > 2$, $n, d \geq 1$, let $\mathfrak{P}^d_{b,n,P}$ be as in Definition 7.4.3 and let $\mathfrak{V}^{d,m_{\mathfrak{U}}}_{b,n,P}$ be defined as above. Then we find*

$$\mathfrak{V}^{d,m_{\mathfrak{U}}}_{b,n,P}(x) = \mathfrak{P}^d_{b,n,P}(x) \text{ for all } x \in \mathfrak{C}^d_b.$$

Proof This can be shown analogously to Proposition 7.4.21, because for each $(u,v) \in \mathfrak{P}^d_{b,n}(P)$ there is exactly one reference vector positioned in u, which is the closest wrt. $m_{\mathfrak{U}}$ for all $x \in u$. □

Corollary 7.4.26 *Let P be some covered logic program, $n \geq 0$, $d \geq 1$, let $\tilde{n}$, $T_{\lfloor P \rfloor_n}$ and $\mathfrak{V}^{d,m}_{b,\tilde{n},P}$ be as defined above for $m \in \{m_{\mathfrak{E}}, m_{\mathsf{m}}, m_{\mathfrak{U}}\}$. Then we find*

$$m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(\mathfrak{V}^{d,m}_{b,\tilde{n},P}(\boldsymbol{\iota}(I)))) \leq b^{-n} \quad \text{for all } I \in \mathcal{I}_{\mathcal{L}}.$$

As in the previous sections, we can now define the corresponding network. Each reference vector is represented by a hidden unit and the weights are set up such that the incoming connections encode the position of the vector and the outgoing connections encode the resulting output.

Definition 7.4.27 *Let P be a covered logic program, $b > 2$, $n \geq 1$ and let $\mathfrak{V}^{d,m}_{b,n}(P)$ be as in Definition 7.4.20 for some metric m. Then we define the corresponding vector-based network $\mathfrak{V}^{d,m}_{b,n,P}$ as follows:*

$$\begin{aligned}
\mathfrak{V}^{d,m}_{b,n,P} &:= \langle m, \mathcal{U}_{\mathsf{inp}}, \omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}}, \mathcal{U}_{\mathsf{hid}}, \omega_{\mathsf{hid}\blacktriangleright\mathsf{out}}, \mathcal{U}_{\mathsf{out}} \rangle \quad \textit{with} \\
\mathcal{U}_{\mathsf{inp}} &:= \{\mathsf{inp}_i(0) \mid 1 \leq i \leq d\} \\
\mathcal{U}_{\mathsf{hid}} &:= \{\mathsf{hid}_i(0) \mid (p,v) \in_i \mathfrak{V}^{d,m}_{b,n}(P)\} \\
\mathcal{U}_{\mathsf{out}} &:= \{\mathsf{out}_i(0) \mid 1 \leq i \leq d\} \\
\omega_{\mathsf{inp}\blacktriangleright\mathsf{hid}} &: \mathcal{U}_{\mathsf{inp}} \times \mathcal{U}_{\mathsf{hid}} \to \mathbb{R} : (\mathsf{inp}_i, \mathsf{hid}_h) \mapsto p^{[i]} \textit{ with } (p,v) \in_h \mathfrak{V}^{d,m}_{b,n}(P) \\
\omega_{\mathsf{hid}\blacktriangleright\mathsf{out}} &: \mathcal{U}_{\mathsf{hid}} \times \mathcal{U}_{\mathsf{out}} \to \mathbb{R} : (\mathsf{hid}_h, \mathsf{out}_o) \mapsto v^{[o]} \textit{ with } (p,v) \in_h \mathfrak{V}^{d,m}_{b,n}(P)
\end{aligned}$$

The corresponding Prolog code is shown in Figure 7.12. First the set of reference vectors is computed. This set of reference vectors is turned into an equivalent network following Definition 7.4.27. The result is shown in Example 7.4.8.

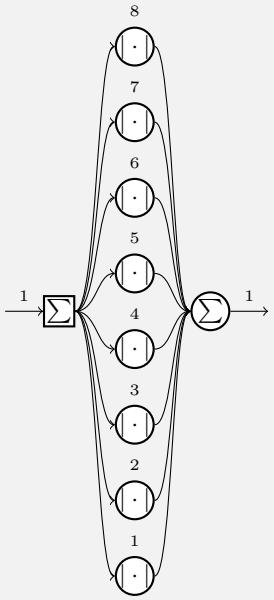
As shown in the following lemma, the approximation $\mathfrak{V}^{d,m}_{b,n,P}$ and the network function of the corresponding $\mathfrak{V}^{d,m}_{b,n,P}$ coincide.

```
vbNetwork(B, Program, LM:D, N, M, Network) :-
    vb(B, Program, LM:D, N, RVs),

    Ui := { unit(I, 0) | I in [1..D]},
    Uh := { unit(I, 0) | P/V in RVs at I },
    Uo := { unit(I, 0) | I in [1..D]},

    I2H := { I/H/W | P/_ in RVs at H, I in [1..D], W in P at I },
    H2O := { H/O/W | _/V in RVs at H, O in [1..D], W in V at I },

    Network = vb(M, Ui, I2H, Uh, H2O, Uo).
```

Figure 7.12: *Implementation to construct an approximating vector based network for a given T_P-operator up to level* `N`*: First, the set of reference vectors is computed using the* `vb/5`*-predicate as shown in Figure 7.11. Afterwards, this set is turned into an equivalent network according to Definition 7.4.27.* `LM : D` *refers to a level mapping of dimension* `D`*.*

Example 7.4.8 An vector based network approximating f_P for our running example. On the left, the call and result of the Prolog implementation is shown, and on the right the resulting network. The hidden layer contains eight units – one for each reference vector. Those units compute the distance to the input signal and the closest is selected as winner. The output layer unit computes the weighted sum of the hidden units outputs, i.e., it receives only the value of the connection from the winner. The approximation is done for level 3.

```
?- Prog = [ e(0)-[],
            e(s(X))-[o(X)],
            o(X)-[not(e(X))] ],
   vbNetwork(4, Prog, eo:1, 3, max, N).

N = vb(max,
    [unit(1,0.00)],
    [1/1/0.003, 1/2/0.018, 1/3/0.065, 1/4/0.081,
     1/5/0.253, 1/6/0.268, 1/7/0.315, 1/8/0.331],
    [unit(1,0.00), unit(2,0.00), unit(3,0.00),
     unit(4,0.00), unit(5,0.00), unit(6,0.00),
     unit(7,0.00), unit(8,0.00)],
    [1/1/0.315, 2/1/0.315, 3/1/0.331, 4/1/0.331,
     5/1/0.253, 6/1/0.253, 7/1/0.268, 8/1/0.268],
    [unit(1,0.00)])
```



Lemma 7.4.28 *Let $f_{\mathfrak{V}}$ be the input-output function of the $\mathfrak{V}^{d,m}_{b,n,P}$ introduced above. Then we find $f_{\mathfrak{V}}(x) = \mathfrak{V}^{d,m}_{b,n,P}(x)$ for all $x \in \mathfrak{U}^d_b$.*

Proof This follows from the above mentioned correspondence between vector based networks and their underlying set of reference vectors. □

We can now give the following theorem, which states the existence of an approximating vector based network for covered logic programs. This theorem states the existence for vector-based networks for the three metrics discussed above. But it can be extended to any metric m satisfying $\mathfrak{V}^{d,m}_{b,n,P}(x) = \mathfrak{P}^d_{b,n,P}(x)$ for all $x \in \mathfrak{C}^d_b$.

Theorem 7.4.29 *Let P be a covered logic program with consequence operator T_P. Let $b > 2$, $n \geq 0$, $d \geq 1$ and let $\tilde{n}$ be as defined above. Then there exists a vector-based network $\mathfrak{V}^{d,m}_{b,\tilde{n},P}$ for $m \in \{m_{\mathfrak{C}}, m_{\mathsf{m}}, m_{\mathfrak{U}}\}$ such that its network function $f_{\mathfrak{V}}$ can be used to compute $T_{\lfloor P \rfloor_n}$, i.e., we find*

$$m_{\mathcal{L},b}(T_{\lfloor P \rfloor_n}(I), \kappa_n(f_{\mathfrak{V}}(\boldsymbol{\iota}(I)))) \leq b^{-n} \quad \text{for all } I \in \mathcal{I}_{\mathcal{L}}.$$

Proof This follows from Lemma 7.4.28 and Corollary 7.4.26 from above. □

This Theorem tells us that there are vector based networks that approximate a given T_P-operator up to a given level of accuracy. And, we can construct such networks using the constructions presented above.

In the following section, we continue with a discussion of the iterated computation. So far, we were concerned with the approximation of a single application of the T_P-operator only. Below, we show under which conditions we can also iterate this approximation to compute the least model of the underlying program.

7.5 Iterating the Approximation

Here, we discuss the iteration of the approximation, i.e., we discuss whether the iterative computation converges to some fixed point. As in Section 3.3, this can be done by recurrently connecting the output layer back to the input layer. Before going into the details, we recall some results from Section 2.3. Under the assumption that a given T_P-operator is contractive, we can use Banach's contraction mapping principle to conclude that the iteration converges to some fixed point $M \in \mathcal{I}_{\mathcal{L}}$. First, we show that the embedded T_P-operator is contractive. We also show that this is not necessarily the case for the approximations discussed above. I.e., we can not guarantee this for all approximating networks constructed. But we present another characterisation, which allows to show that the iteration converges nonetheless.

7.5.1 Contractivity of the Approximation

Lemma 2.3.19 in Section 2.3 shows that the Lipschitz constant L_T of a contractive T_P-operator is bounded by $L_T \leq \frac{1}{b}$. Together with Lemma 7.3.5 we obtain the following corollary:

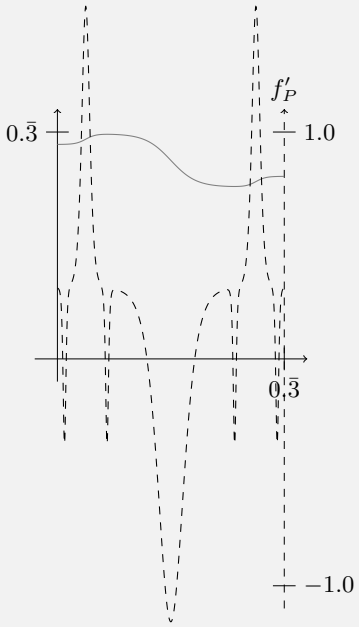
Corollary 7.5.1 *Let T_P be a contractive mapping on $(\mathcal{I}_{\mathcal{L}}, m_{\mathcal{L},b})$ wrt. a bijective level mapping, let $b > 3$ and let f_P be as defined above. Then f_P is contractive on $(\mathfrak{C}^d_b, m_{\mathsf{m}})$ with Lipschitz constant $L_f = \frac{1}{b-2}$.*

To put it another way, not only T_P is contractive, but also the embedded version f_P, even thought it is not as contractive as T_P, i.e., with a larger associated Lipschitz constant. This gives immediate rise to the following theorem, which follows again from Banach's theorem. The contractivity of a linear extension $\bar{f_P}$ of the embedded function f_P was first shown in [HKS99]. The authors showed it directly by using their definition of $\bar{f_P}$ for $b = 4$.

Theorem 7.5.2 *Let T_P be a contractive mapping on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ wrt. a bijective level mapping and let M be its least fixed point. Let $b > 3$ and let f_P be as defined above. Then the sequence $x, f_P(x), f_P^2(x), \ldots$ converges for all $x \in \mathfrak{C}_b^d$ to some fixed point m such that $\boldsymbol{\iota}(M) = m$.*

Before trying to prove that our approximations are contractive, we look at the derivative of the sigmoidal version. Example 7.5.1 shows the plot of the sigmoidal approximation and the corresponding derivative. A mapping is contractive if and only if the absolute value of the corresponding derivative is below 1 for all inputs. Obviously this approximation is not contractive, because its derivative has absolute values greater than 1.

Example 7.5.1 A sigmoidal approximation of f_P as in Example 7.4.3 but for level 4 together with the corresponding derivative. Please note that the derivative is scaled by $\frac{1}{3}$ on the ordinate and plotted as dashed line against the right axis.



The problem becomes even more obvious, by looking at Figure 7.13. Let x_a, x_b, x_0, y_0 as depicted in Figure 7.13. If the truth value of some atom A of level $n+1$ depends on some atom of level n, we find

$$x_a = x_0 + \frac{b^{-n}}{b-1} \cdot \frac{1}{b} \qquad x_b = x_0 + \frac{b^{-n}}{b-1} \cdot \frac{b-1}{b}$$
$$f(x_a) = y_0 \qquad f(x_b) = y_0 + b^{-(n+1)}$$

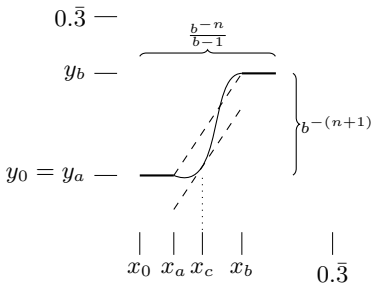


Figure 7.13: *Approximation based on the constant pieces are not necessarily contractive if the underlying f_P is contractive: The figure shows the case that the truth value of some atom of level $n+1$ depends directly on some atom of level n. Then, all resulting approximations based on the constant pieces are non-contractive.*

From the mean-value theorem, we can conclude that there must be some $x_c \in [x_a, x_b]$ such that

$$\begin{aligned} f'(x_c) &= \frac{f(x_b) - f(x_a)}{x_b - x_a} \\ &= \frac{y_0 + b^{-(n+1)} - y_0}{(x_0 + \frac{b^{-n}}{b-1} \cdot \frac{b-1}{b}) - (x_0 + \frac{b^{-n}}{b-1} \cdot \frac{1}{b})} = \frac{b^{-n} \cdot \frac{1}{b}}{\frac{b^{-n}}{b-1} \cdot \frac{b-1}{b} - \frac{b^{-n}}{b-1} \cdot \frac{1}{b}} = \frac{1}{1 - \frac{1}{b-1}} \\ &= \frac{b-1}{b-2} > 1 \end{aligned}$$

I.e., we can not built a contractive approximations based on the constant pieces constructed above. Therefore, we abandon this approach and continue with a new characterisation.

7.5.2 Convergence of the Approximations wrt. Hyper Squares

We now pursue a different approach to show the convergence of our approximations. It is based on the following observation:

Lemma 7.5.3 *Let T_P be contractive on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ and let M be its fixed point. Then there exists a sequence $F_0 \supset F_1 \supset F_2 \supset \ldots$ with $F_i \in \Omega^i\left(\mathfrak{U}_b^d\right)$ and $\boldsymbol{\iota}(M) \in F_i$ for all $i \geq 0$.*

Proof This follows directly from Lemma 7.2.12 and the fact that $\boldsymbol{\iota}(M) \in \mathfrak{C}_b^d$. □

Using this sequence of F_i, we can define another sufficient condition to guarantee convergence of approximations. We require the image of some F_i from Lemma 7.5.3 to be contained in F_{i+1}. By definition, we find $F_i \supset F_{i+1}$ and thus the sequence $x, f(x), f^2(x), \ldots$ to converge to $\boldsymbol{\iota}(M)$ as shown in Proposition 7.5.5 below.

Definition 7.5.4 *Let T_P be contractive on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$, let f_P be its d-dimensional embedding and let $F_0 \supset F_1 \supset F_2 \supset \ldots$ be defined as in Lemma 7.5.3. We call an approximation $f : \mathbb{R}^d \to \mathbb{R}^d$ of f_P F-*converging *if $f(F_i) \subseteq F_{i+1}$ for all $i \geq 0$.*

Proposition 7.5.5 *Let T_P be contractive on $(\mathcal{I}_\mathcal{L}, m_{\mathcal{L},b})$ with fixed point M and let f be an F-converging approximation of f_P. Then we find the sequence $x, f(x), f^2(x), \ldots$ to converge to $\boldsymbol{\iota}(M)$ for all $x \in \mathfrak{U}_b^d$.*

Proof To show the convergence, we compute some n for each $\varepsilon > 0$ such that for all $i \geq n$ we find $m_{\mathsf{m}}(f^i(x), \boldsymbol{\iota}(M)) \leq \varepsilon$. From Definition 7.5.4 follows that $f^i(x) \in F_i$ for all $x \in \mathfrak{U}_b^d$. Using the Definition of hyper cube of level n, we know that $m_{\mathsf{m}}(x_1, x_2) \leq \frac{b^{-i}}{b-1}$ for all $x_1, x_2 \in u \in \Omega^i\left(\mathfrak{U}_b^d\right)$. Therefore, we can conclude $m_{\mathsf{m}}(f^i(x), \boldsymbol{\iota}(M)) \leq \frac{b^{-i}}{b-1}$. Using some $n \geq \log_b(\varepsilon(b-1))$, we find $m_{\mathsf{m}}(f^i(x), \boldsymbol{\iota}(M)) \leq \varepsilon$, which end the proof. □

I.e., if an approximation is F-converging, we can use it to approximate not only T_P itself, but also its least fixed point M. But in the constructions presented above, we have only been concerned with the approximation up to some level n. Hence, we are not able to show F-convergence. Therefore, we soften the convergence condition a little by requiring it for $0 \leq i < n$ only. This is enough to ensure convergence up to level n.

Definition 7.5.6 *Let T_P be contractive on $(\mathcal{I}_{\mathcal{L}}, m_{\mathcal{L},b})$, $n \geq 0$ and let f_P, d, M and F_i be as in Definition 7.5.4. We call an approximation $f : \mathbb{R}^d \to \mathbb{R}^d$* F-converging up to level n *if $f(F_i) \subseteq F_{i+1}$ holds for all $0 \leq i < n$.*

Now, we show that this condition is sufficient to guarantee the convergence up to the level n. I.e., if an approximation is F-converging up to level n, then we can give a bound of the error after at least n iterations, because we know that the n-th image is contained in the same hyper-square of level n as the embedded model. Hence the distance is bounded by the size of this hyper-square.

Proposition 7.5.7 *Let T_P be contractive on $(\mathcal{I}_{\mathcal{L}}, m_{\mathcal{L},b})$ with fixed point M. Let f be an approximation of the embedding f_P and let it be F-converging up to level n. Then we find $m_{\mathsf{m}}(f^i(x), \boldsymbol{\iota}(M)) \leq \frac{b^{-n}}{b-1}$ for all $i \geq n$ and all $x \in \mathfrak{U}_b^d$.*

Proof By definition, we find $f^i(x) \in F_n$ for all $i \geq n$. Furthermore, we know that $m_{\mathsf{m}}(x_1, x_2) \leq \frac{b^{-n}}{b-1}$ $x_1, x_2 \in u \in \Omega^i\left(\mathfrak{U}_b^d\right)$, which allows to conclude $m_{\mathsf{m}}(f^i(x), \boldsymbol{\iota}(M)) \leq \frac{b^{-n}}{b-1}$ for all $i \geq n$ and all $x \in \mathfrak{U}_b^d$. □

It remains to be shown that our approximations are indeed F-converging up to some level n. Unfortunately, we can do this for the sigmoidal and the vector-based approximations only. The approximation by raised cosines is not converging.

Before showing F-convergence for our approximations, we give the following lemma to simplify the proofs below. The lemma introduces the following condition on the approximations f of $\mathfrak{P}_{b,n,P}^d$ corresponding to program which is acyclic wrt. $m_{\mathcal{L},b}$: Both functions must map all hyper-squares into the same hyper-squares. For those programs we find that the image of some hyper-square u of level i wrt. $\mathfrak{P}_{b,n,P}^d$ is contained in a hyper-square of level $i+1$. This follows from the contractivity of the underlying T_P-operator. If the image of f is also contained in this hyper-square, we can conclude that f is F-converging up to level n.

Lemma 7.5.8 *Let P be acyclic wrt. $m_{\mathcal{L},b}$. Let $d \geq 1$, $n \geq 0$, $\mathfrak{P}_{b,n,P}^d : \mathfrak{C}_b^d \to \mathbb{R}^d$ and ε_n be as in Definition 7.4.3. Let $f : \mathbb{R}^d \to \mathbb{R}^d$ be an approximation of $\mathfrak{P}_{b,n,P}^d$ such that $m_{\mathsf{m}}\left(f(x), \mathfrak{P}_{b,n,P}^d(x)\right) \leq \varepsilon_n$. Then, f is F-converging up to level n.*

Proof We use F_i as defined in Lemma 7.5.3, i.e., $F_0 \supset F_1 \supset F_2 \supset \ldots$ with $F_i \in \Omega^i\left(\mathfrak{U}_b^d\right)$ and $\boldsymbol{\iota}(M) \in F_i$ for all $i \geq 0$. From P being acyclic wrt. $m_{\mathcal{L},b}$, we can conclude that $\cup_{x \in F_i \cap \mathfrak{C}_b^d} \mathfrak{P}_{b,n,P}^d(x) \subseteq F_{i+1}$ holds for all $0 \leq i < n$. This follows from the fact that every

iteration of a contractive T_P fixes at least one level in the output. Because $\mathfrak{P}^d_{b,n,P}$ is defined to output a value in the centre of the output-hyper square of level n and it follows from $m_{\mathsf{m}}\left(f(x), \mathfrak{P}^d_{b,n,P}(x)\right) \leq \varepsilon_n$ that f is F-converging up to level n. □

The following corollaries follow from this lemma together with Lemma 7.4.6, 7.4.9 and 7.4.25, respectively. They state that the approximations by step functions, sigmoidals and the vector-based approximation fulfil this condition and are hence F-converging up to level n.

Corollary 7.5.9 *Let P be acyclic wrt. $m_{\mathcal{L},b}$. Then $\mathfrak{T}_{b,n,P}$, $\mathfrak{S}_{b,n,P}$ and $\mathfrak{V}^{d,m}_{b,n,P}$ as introduced in Definition 7.4.5, 7.4.8 and 7.4.20 respectively are F-converging up to level n.*

As mentioned above, the approximation by raised cosines is not converging. This can easily be seen by looking at the plot from Example 7.4.5. The problem is that the function drops back to 0 between two neighbouring constant pieces. Corollary 7.5.9 state that the sigmoidal and vector-based approximations are F-converging up to level n. Therefore, we can use them to approximate not only T_P, but also its least model M as stated in the following theorem.

Theorem 7.5.10 *Let P be acyclic wrt. $m_{\mathcal{L},b}$ and let M be the fixed point of the associated T_P-operator. Then there exist recurrent sigmoidal and vector-based networks such that the difference between the networks output and $\boldsymbol{\iota}(M)$ after at least n iterations is $\leq \frac{b^{-n}}{b-1}$ if the network is initialised to a value within $\mathfrak{U}^d_b$.*

Proof We can turn the sigmoidal and the vector-based networks constructed above into recurrent ones by connecting the output layer to the input layer, which leads to an iterated computation of the corresponding network functions. This observation together with Corollary 7.5.9 and Proposition 7.5.5 ends the proof. □

This last theorem is a constructive generalisation of Theorem 4 in [HKS99]. There the authors showed it for acyclic programs and sigmoidal networks. But as already mentioned above, this proof was existential and there was no algorithm to construct the approximating networks. The version presented here thus generalises the old result, by enlarging the class or programs and by incorporating other network types as well.

7.6 Vector-Based Learning on Embedded Interpretations

In this section, we describe the adaptation of the vector based networks from Section 7.4.4 during training. I.e., how the weights and the structure of a network are changed, given training samples with input and desired output. This process can be used to refine a network resulting from an incorrect or incomplete program. It can also be used to train a network completely from scratch, i.e., using an initial network with a single hidden layer unit.

First we discuss the adaptation of the weights and then the adaptation of the structure by adding and removing hidden-layer units. Some of the methods used here are adaptations of ideas described in [Fri98]. In particular, we used the notion of utility as known from vector-based neural networks.

This type of training was first described in [BHHW07] where we also presented preliminary results which serve as a proof of concept.

7.6.1 Adapting the Weights

As usual for supervised training of neural networks, we assume pairs of input and output vectors as training samples. Let $\mathbf{x}$ be some input vector, $\mathbf{y}$ be the desired output and u be the winner-unit within the hidden layer. Let $\mathbf{w}_{in} = (w_1, \ldots w_d)$ with $w_i = \omega_{\mathsf{inp}_i \blacktriangleright \mathsf{hid}}$ denote the weights of the incoming connections of u and $\mathbf{w}_{out} = (v_1, \ldots v_d)$ with $v_i = \omega_{\mathsf{hid} \blacktriangleright \mathsf{out}_i}$ be the weights of the outgoing connections. To adapt the system, we move u towards the centre $\mathbf{c}$ of the smallest hyper-square containing both, the unit u and the input vector $\mathbf{x}$.

$$\mathbf{w}_{in} \leftarrow \mu \cdot \mathbf{c} + (1 - \mu) \cdot \mathbf{w}_{in}.$$

The output of the system is changed towards the desired output by adapting the outgoing weights:

$$\mathbf{w}_{out} \leftarrow \eta \cdot \mathbf{y} + (1 - \eta) \cdot \mathbf{w}_{out}.$$

η and μ are predefined learning rates. Note that (in contrast to the methods described in [Fri98]) the winner unit is not moved towards the input, but towards the center of the smallest hyper-square including the unit and the input. The intention is that units should be positioned in the centre of the hyper-square for which they are responsible. Figure 7.14 depicts the adaptation of the incoming connections in two different scenarios.

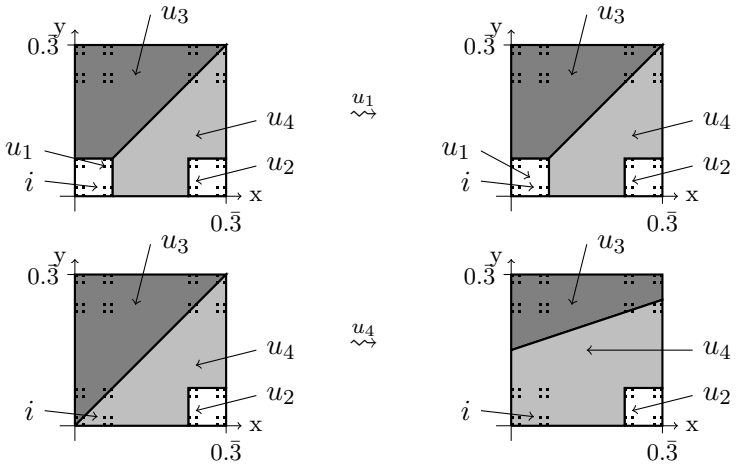


Figure 7.14: *The adaptation of the input weights for a given input i: The first row shows the result of adapting u_1. The second row shows the result if u_1 was not there. In this case u_4 would be selected as winner. To emphasise the effect, we used a learning rate $\mu = 1.0$, i.e., the winning unit is moved directly into the centre of the hyper-square.*

7.6.2 Adding New Units

The adjustment described above enables a certain kind of expansion of responsible areas within the network by allowing units to move to positions where they are responsible for larger areas of the input space. A refinement should take care of densifying the network in areas where a great error is caused. Every unit accumulates the error for those training samples, for which it is the winner. If this accumulated error exceeds a given threshold, the unit is selected for refinement. I.e., we try to figure out the area it is responsible for and a suitable position to add a new unit.

Let u be a unit selected for refinement. If it occupies a hyper-square on its own, then the largest such hyper-square is considered to be u's responsibility area. Otherwise, we take the

smallest[2] hyper-square containing u. Now u is moved to the centre of this area. Information gathered by u during the training process is used to determine a sub-hyper-square into whose centre a new unit is placed, and to set up the output weights for the new unit. All units collect statistics to guide the refinement process. E.g., the error per sub-hyper-square or the average direction between the centre of the hyper-square and the training samples contributing to the error could be used (weighted by the error). This process is depicted in Figure 7.15.

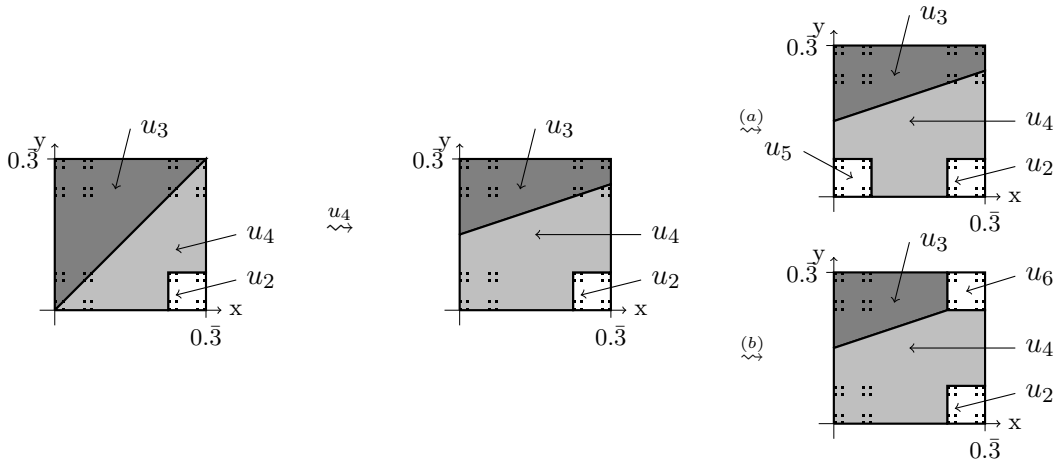


Figure 7.15: *Adding a new unit to support u_4: First, u_4 is moved to the centre of the hyper-square it is responsible for. There are four possible sub-hyper-squares to add a new unit. Because u_4 is neither responsible for the north-western, nor for the south-eastern sub-hyper-square, there are two cases left. If most error was caused in the south western sub-hyper-square (a), a new unit u_5 is added there. If most error was caused in the north-eastern area (b), a new unit u_6 is added there.*

7.6.3 Removing Inutile Units

Each unit maintains a utility value, initially set to 1, which decreases over time and increases only if the unit contributes to the network's output. The contribution of a unit is the expected increase of error if the unit is removed [Fri98]. If it drops below a threshold, the unit gets removed as depicted in Figure 7.16.

The methods described above, i.e., the adaptation of the weights, the addition of new units and the removal of inutile ones, allows the network to learn from examples. While the idea of growing and shrinking the network using utility values, was taken from vector-based networks [Fri98], the adaptation of the weights and the positioning of new units are specifically tailored for the type of function we like to represent, namely functions on $\mathfrak{U}_b^d$. The preliminary experiments described below show that our method actually works.

7.7 Summary

In this chapter we have studied the embedding of first-order programs into connectionist systems. After repeating the existential results first presented in [HKS99], we discussed construc-

[2]Please note that the smallest hyper-square does not necessarily exists, i.e., if tow units are positioned exactly on one element of $\mathfrak{C}_b^d$. Therefore, we use an upper limit for the level while looking for this hyper-square in our implementation.

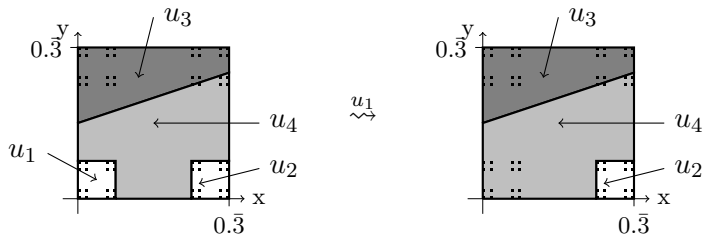


Figure 7.16: *Removing an inutile unit. Let us assume that the outgoing weights of u_1 and u_4 are equal. In this case we would find that the over-all error would not increase if we remove u_1. Therefore its utility would decrease over time until it drops below the threshold and the unit is removed.*

tions in great detail. First, we have shown how to embed interpretations and the semantic operators into the real numbers. Then, we showed how to approximate this operator using a piecewise constant function. This piecewise constant function was the basis for the connectionist implementations presented thereafter. In Section 7.4.2 we have shown how to construct three layer feed-forward networks with sigmoidal units in the hidden layer. Afterwards, we have presented approaches using RBF networks of raised cosines and vector-based networks in Section 7.4.3 and 7.4.4, respectively. We have discussed the iteration of the approximation and gave conditions under which the iterated application of the approximating functions converge to the least fixed point of the embedded T_P-operator. Finally, we have discussed a new training scheme for the vector-bases networks.

We are now in a position to choose between different network architectures. The sigmoidal and RBF approach presented above have still a limited precision due to the underlying 1-dimensional embedding. But the vector-based approach presented in Section 7.4.4 can be used to approximate semantic operators with arbitrary precision. Because we can apply the multi-dimensional embedding, the precision is only limited by the available memory. It can be improved by increasing the number of dimensions used in the embedding and by increasing the number of hidden units to obtain a finer partition.

8 Conclusions

... where we review the results achieved so far and discuss possible further work. We start with a summary of the results achieved in this thesis. Afterwards, we review the challenge problems discussed in Section 1.4. Because some of those problems turn out to be answered – at least partially. Finally, we discuss possible future research directions to advance the field of neural-symbolic integration.

8.1 Summary

This section provides a summary of the main results achieved in this thesis.

8.1.1 Chapter 1 & 2: Introduction and Preliminaries

In Chapter 1, the motivation and the state of the art in the field of neural-symbolic integration have been presented. We discussed the general approach followed in most neural-symbolic systems. This approach can be depicted as the neural-symbolic cycle, shown in Figure 1.1 on page 4. The integration of a symbolic and a connectionist part is achieved by first embedding symbolic knowledge into a connectionist system, which can then be refined using standard learning algorithms. The refined knowledge can later be extracted to provide a (human) readable version of the acquired rules. We have also discussed a classification scheme for systems in the field and identified nine challenging problems that need to be answered.

All necessary notions and results for later parts have been introduced in Chapter 2. In particular, we reviewed basic results of:

- metric spaces, continuous and contractive functions and so-called iterated function systems,
- logic programs and their associated semantic operators,
- binary decision diagrams as a representation of propositional formulae, and
- connectionist systems (in particular 3-layer perceptrons, radial basis function networks and vector based networks)

8.1.2 Chapter 3 to 6: The Propositional Case

Propositional neural-symbolic systems have been discussed in Chapter 3 to 6. Starting from well known results concerning the embedding of propositional rules into connectionist systems, we discussed the training of those systems with respect to symbolic background knowledge and the extraction of the acquired knowledge afterwards. Furthermore, we discuss an application of the ideas to the problem of part of speech tagging in Chapter 4.

Embedding Propositional Rules into Connectionist Systems has been discussed in Chapter 3. We repeated some well known results concerning the representation of semantic operators for propositional logic programs. This has been done using a unifying notation which covers the results of [HK94] and [dGZ99]. This new representation does hopefully lead to a better understanding of the ideas, because all constructions are presented on a slightly more abstract level hiding the obscuring numerical details. In addition, Prolog implementations for all algorithms have been provided.

We showed that the semantic operator T_P associated to a logic program P can be computed within 3-layer networks composed of threshold units. Then, we have extended the result to sigmoidal networks, i.e., to networks composed of non-threshold units. In this case the proof of the representation result is slightly more involved. But due to the new approach of the characterisation, it might be easier to consume than in previous papers. Finally, we discussed the repeated application of the operator. As already mentioned above, the results presented in this chapter have been achieved before, but the new formalisation provides a more unified view on the area and the algorithms.

An Application to Part of Speech Tagging has been discussed in Chapter 4. We showed how to embed grammatical knowledge into a connectionist system along the lines described in Chapter 3. Furthermore, we showed in experiments that this embedding has a positive impact on the training of the resulting network. To the best of our knowledge, this is the biggest problem ever tackled using a neural-symbolic approach.

Connectionist Learning and Propositional Background Knowledge has been discussed in Chapter 5. During experiments using the embedding discussed before, we observed that:

- very general rules are acquired quickly during the training process,
- very specific rules are overwritten during the first steps of the training, and
- the errors made by a network can be analysed to obtain correcting rules, which can then be embedded.

Those observations led to a the rule insertion cycle shown in Figure 5.1 on page 78. In this chapter we have discussed the general scheme for the repeated insertion of rules during the training process and showed that the approach does actually work for a simple application. We found that a connectionist classifier obtains better results if error-correcting rules are repeatedly embedded during the training. The rules have been obtained through an analysis of the behaviour of the network. Even though a very naive algorithm has been used to construct rules, the network's performance increased.

Extracting Propositional Rules from Connectionist Systems has been discussed in Chapter 6. After defining the rule-extraction problem formally, we have discussed a new decompositional approach to solve it.

The introduced CoOp-approach decomposes the network into its basic building blocks, so-called perceptrons. From those perceptrons, we extracted binary decision diagrams representing conditions which turn the perceptron inactive or active. The construction was presented using search trees, which have later been pruned and finally been turned into decision diagrams. After re-composing the intermediate results, we showed that the obtained reduced ordered binary decision diagram is indeed a solution to the rule extraction problem.

We have furthermore studied the incorporation of integrity constraints into the extraction process. Integrity constraints are usually applied to the extraction result afterwards, to shrink its size. The CoOp-approach presented here allows a straight-forward integration during the extraction. This leads to far smaller intermediate results.

Because the extracted binary decision diagrams can easily be turned into logic programs, Chapter 3 to 6 contain the full neural-symbolic cycle for logic programs and feed-forward neural networks.

8.1.3 Chapter 7: The First-Order Case

In Chapter 7 we have discussed the embedding of first-order rules into connectionist systems. After repeating the existential result from [HKS99], we discussed the embedding of first-order interpretations into real vectors. For this, we introduced multi-dimensional embeddings. Using this embedding, we obtained the resulting space of embedded interpretations, which turned out to be a multi-dimensional version of the Cantor-set. The intermediate results of a particular iterated function system constructing this set – called hyper squares – play an important role during the construction of approximations.

Afterwards, we have studied the embedding of the associated consequence operators and discussed some metric properties of it, which allow an approximation using connectionist systems. The approximation has been realised using

- piecewise constant functions,
- sigmoidal networks,
- radial basis function networks, and
- vector-based networks.

After proving all those approximation results, we discussed the iteration of the approximation. Unfortunately, it turns out that Banach's contraction mapping theorem, which is usually used to prove convergence, can not be used here. To show the convergence of the iterated application of the embedded operator, we introduced the notion of convergence with respect to the above mentioned hyper squares. This special characterisation of approximation can be used to conclude that all but the RBF approximation converge to the fixed point of the embedded operator.

Finally, we have discussed the training for vector-based networks, because it is possible to incorporate some properties of the underlying set of embedded interpretations.

8.2 Challenge Problems Revised

In Section 1.4 we have discussed nine challenge problems which need to be solved to create a fully integrated system for (first-order) reasoning within a connectionist system. Because some of them have at least partially answered in this thesis, we re-discuss those problems here – now from a more informed point of view.

8.2.1 How Can (First-Order) Terms be Represented in a Connectionist System?

In Chapter 7 we have presented an approach to answer this question. We used a multi-dimensional embedding of first-order interpretations into vectors of real numbers. This has been done using so called level mappings, which assign natural numbers to ground atoms. Using those numbers we can map ground atoms as well as sets of ground atoms to vectors of real numbers. The level mapping is used to specify the importance of atoms. This importance gives rise to the following notion of approximation underlying the presented approach: An interpretation is approximated the better the more important atoms are mapped to the correct truth value by the network.

Even though the presented approach might not be the ultimate answer to the question, we could show that it is at least *an* answer to it. Furthermore, it has some interesting properties: The accuracy of the network does only depend on the available hardware resources. Spending more memory directly improves the quality of the approximation. Unlike in phase coding mechanisms, where only finitely many atoms can be represented, we can at least approximate infinitely many.

Nonetheless, the question is not yet answered in a completely satisfying way. The embedding is able to embed ground atoms only. Thereby, the variable binding problem has been circumvented. But at the same time, the rule extraction problem has been made more difficult. Due to the distributed embedding, it is very hard to recognise structure, which would be necessary to extract first-order rules from a trained network. This issue is further discussed in the following section.

8.2.2 Can (First-Order) Rules be Extracted from a Connectionist System?

In Chapter 6, we have discussed a new propositional approach to extract rules. The preliminary implementation prevented an application to real world problems, but nonetheless we have seen that the pruning methods are effective. Therefore, we may hope that the method is applicable to larger problems, and maybe also to problems which are not solvable using other systems.

Even though there is some progress for propositional rule extraction, it is completely unclear yet how to extract first-order rules from a connectionist system. As mentioned above, we can embed first-order rules into a network in an approximative way. But the underlying grounding hides possibly emerging structural patterns in lots of instances of rules. This makes it very hard, if not impossible, to find any pattern at all. Therefore, we can only conclude that this challenge problem remains open for the future.

8.2.3 Can Neural Knowledge Representation be Understood Symbolically?

This question relies very much on an answer of the previous two. We can not really hope to understand connectionist representations if there is no way of embedding and extracting symbolic rules into and from a network.

For the propositional case, we might be able to study the evolution of rules during the training process, by continuously tracking it. This should provide some insights into the connectionist knowledge acquisition and modification. As mentioned above, this experiment can not even be done for the first-order case due to the missing extraction method.

A different approach to answer this question could rely on the method introduced in Chapter 5. There we have shown that a repeated insertion of error-correcting rules into the network is possible. Using the underlying idea of analysing the errors might also yield new insights into the connectionist processes. We could study the evolution of the errors which should yield a better understanding of the connectionist learning and thus of the connectionist knowledge representation.

But both experiments have not been performed so far, and thus this question does also remain to be answered.

8.2.4 Can Learning Algorithms be Combined with Symbolic Knowledge?

The positive influence of symbolic rules on connectionist training algorithms has successfully been shown in Chapter 5. The proposed method consists of a repeated analysis of the errors made by the network and the insertion of correcting rules. This produces improved results on a simple classification task. Even though the presented approach is only a starting point for further research it shows that established learning algorithms can indeed be influenced using symbolic background knowledge.

8.2.5 Can Multiple Instances of First-Order Rules be Used in a Connectionist System?

As already mentioned in Section 1.4, there exist different solutions to the problem. E.g., phase coding mechanisms have been used, the maximal number of available instances have been restricted or in the approach from Chapter 7, the problem has just been circumvented.

Nonetheless, the question remains open, because none of the approaches is satisfying yet. Restricting the number of instances turns the system necessarily incomplete. Circumventing the problem as done above, does not really solve it but hides it within the approximation. The better the approximation the more (ground) instances of rules are available to the system. Hence, the question needs to be further investigated.

8.2.6 Can Insights from Neuroscience be Used to Design Better Systems?

This question has not been addressed within this thesis. Hence, it is as open as before from a purely technical point of view as pursued here. But e.g., Tianming Yang and Michael N. Shadlen showed in [YS07] that monkeys can be trained to perform some form of probabilistic reasoning. And moreover, that certain neurons implement addition and subtraction of probabilities. Even though, they could not show how higher-level cognitive reasoning is performed on a neural base, their results show nonetheless that a symbolic interpretation of single neurons is possible. This result may not immediately lead to a better neural-symbolic system but it indicates that the research in this area is going into the right direction.

8.2.7 Can the Relation Between Neural-Symbolic and Fractal Systems be Exploited?

This question has been raised by the findings from [Bad03, BH04] as well as [BDJ+99]. We did not advance the state of the art directly, but the constructions presented in Chapter 7 provide the foundations for a multi-dimensional investigation. In [Bad03, BH04], the relation between the embedded T_P-operator and iterated function systems has been discussed for the one-dimensional case only. Using the multi-dimensional embedding presented here, we may lift the earlier results to a more general setting.

8.2.8 What does a Theory for the Neural-Symbolic Integration Look Like?

This thesis contains a number of steps towards a unifying theory:

- The classification scheme presented in Secion 1.3,
- the uniform treatment of the embedding for propositional rules discussed in Chapter 3, and
- the constructions from Chapter 7, which incrementally built upon each other.

The classification scheme was intended to bring some order into the field and, thus, provides a basis for a unified view of the subject. By classifying the different existing approaches, we might find similarities, contrasts and missing pieces. Once all existing approaches have been classified, the bigger picture should be clearer. For this, we might possibly need to extend the scheme but it should still provide a nice starting point.

Another step towards a unifying theory has been taken in Chapter 3, where we discussed the embedding of propositional rules into connectionist systems. The constructions have been presented on a more abstract level then previously. This allows to compare at least a number of different approaches.

The constructions presented in Chapter 7, which build upon each other, constitute another step towards such a theory. Starting with the most simple case of threshold functions, we have discussed different network architecture. The relation between all of them is quite clear and we are now in a position to choose a target architecture depending on our needs.

8.2.9 Can Neural-Symbolic Systems Outperform Conventional Approaches?

Already the very first neural-symbolic systems could outperform purely symbolic and purely connectionist approaches on simple application domains [TS94]. But the tackled problems have been relatively small in the number of training data and the complexity of the embedded rules. First experiments performed in [MBRH07] indicate that neural-symbolic systems can also be used to solve real world problems. We could show that the integration of symbolic

background knowledge improves the performance of a neural part of speech tagger. The performed experiments have been magnitudes bigger than previous ones.

Nonetheless, the question must remain open until other applications are found. The issue of application areas and benchmark problems is further discussed below.

8.3 Further Work

Further extensions in the area of neural-symbolic integration as pursued here, can be grouped into five sub categories:

1. Possible Applications Domains
2. Possible Extensions
3. Expansion and Unification of the Theory
4. The Relation to Neuro- and Cognitive Science
5. Symbolic-Neural Systems

After describing those issues in more detail we discuss open problems which are directly related to the technical content of this thesis in Section 8.3.6.

8.3.1 Possible Applications Domains

The identification of possible application domains is one of the most important issues to be addressed in the near future. More applications are needed, and in particular we need to identify domains in which neural-symbolic systems outperform conventional approaches. Some possible candidates are

- Natural Language processing,
- Ontology learning, and
- Cognitive robotics.

But we do not only need to define application domains and provide case studies, we should also try to collect benchmark-problems, analogously to [AN07]. Such a collection would allow a better comparison among neural-symbolic approaches and with conventional methods.

Natural Language Processing

A very promising application area is natural language processing. Already back in 1990, Jeff L. Elman showed that recurrent neural networks can be used to detect temporal patterns in language [Elm90]. This result triggered the research in natural language processing using neural networks. The part of speech tagging problem [vH99] is another example of the successful application of neural networks in this area. State of the art taggers are either based on explicit statistical models capturing the internal structure of sentences, or on neural networks trained to solve the task [Sch94]. In Chapter 4 and [MBRH07], we could also show that the accuracy of neural networks can be increased using a neural-symbolic approach. The domain of natural language processing is interesting in this area, because huge amounts of symbolic knowledge is already available in form of grammars or statistical rules (at least for most languages). Researchers of either area are competing on a very high level. I.e., so far the rule-based and the connectionist approaches are equally good, but as always for either system there are domains

where it outperforms the other. This gives rise to the hope that the integration of both approaches can indeed lead to better systems and first results encourage further research in this direction.

Ontology Learning

In [HBdG05], we have argued that ontology learning could be a fruitful application domain for neural-symbolic systems. Semantic technologies are advancing very rapidly. But nonetheless, most of today's internet is accessible for humans only. The information is presented textually, and no annotations are provided which are needed for a semantic processing of the information. The construction of ontologies is the current bottleneck. So far, ontologies are hand-crafted due to the fact that the textual representations of information is not usable for machines.

As argued above, natural language processing is a candidate area in which neural-symbolic systems may outperform conventional approaches. Starting from there, we could develop a neural-symbolic ontology learner, which bridges the gap between huge amounts of textual information and semantic technologies, which require knowledge in some symbolic form.

Cognitive Robotics

In [BDT05] the authors propose a neural-symbolic behaviour language to specify the behaviours for reactive and deliberative robots. The high-level specifications are compiled into a connectionist system which is then implemented using FPGAs. The application domain of cognitive robotics seems to be promising, because its problems are the high-level objectives of neural-symbolic integration. I.e., the design of adaptive systems which can be initialised with symbolic background knowledge and which are able to adapt to new situations and environments.

Benchmarks are Needed!

We need benchmark problems to compare different approaches and to study their relationship. Even though a classification scheme as introduced in Section 1.3 helps to establish an order, the classification of the different approaches is rather subjective in some dimensions. Benchmark problems might provide the basis for an experimental evaluation of the approaches and a detailed comparison. This investigation should yield some new insights into the area and help to design better systems.

8.3.2 Possible Extensions

Possible extension for the state of the art in neural-symbolic integration include among many other directions:

- further advancement for learning symbolic knowledge within connectionist systems,
- the connectionist implementation of other inference mechanisms, and
- the extension to other (non-classical) logics.

Learning Symbolic Knowledge

A first result on the incorporation of symbolic knowledge into the training process has been presented in Chapter 5. We discussed a general scheme, the rule-insertion cycle, which allows the usage of symbolic rules and background knowledge during the training of a connectionist

system. But as mentioned there, the presented results mark only the starting point for further investigations.

One sub-problem of the learning in neural-symbolic systems is the automatic construction of the embedding function. Throughout this thesis, the embedding has been assumed to be fixed. But this is not a necessary condition as shown in the RAAM system [Spe94a] and by Helmar Gust and Kai-Uwe Kühnberger in [GK05]. In both approaches, the embedding is learned by the network during the training process.

Powerful Inference Mechanisms

Throughout this thesis, we have been concerned with model generation by iterating the semantic operator associated to a logic program. But of course other inference methods are possible. E.g., in the approach presented in [GK05], the network learns representations of logical formulae such that (logically) equivalent formulae are mapped to identical representations. This allows query answering by embedding the query and comparing it to the representation of a tautology. If both coincide, the query is a tautology as well and hence true.

The SHRUTI system as discussed in Section 1.2.6, implements so-called reflexive reasoning. I.e., the form of reasoning performed by humans which yields answers very quickly for certain queries.

These are just two other approaches which shall serve as examples for other possible inference mechanisms. Actually, this is a point where insights from cognitive and neuroscience might be incorporated. If a certain form of reasoning is identified to be performed by humans, we should try to come up with a neural-symbolic implementation of it.

Non-Classical Logics

A number of non-classical extensions have been studied in [dGGL05a]. These include modal, temporal and intuitionistic logics. The study of such logics in the context of neural-symbolic integration is necessary, because it is widely accepted that only such logics may be used to model human thinking. Keith Stenning and Michiel van Lambalgen show in [SvL08] that completed logic programs under a three-valued Fitting semantic can indeed model human reasoning. A connectionist implementation of their ideas can be found in [HR09b, HR09a].

8.3.3 Expansion and Unification of the Theory

Besides extending the results, researches in the field should try to create a unifying theory. As mentioned above, such a theory underlying all approaches in the area is still missing. This theory, should enable a better comparison of the approaches and their relation.

Small steps towards such a theory have been taken here. Chapter 3 and 7 present a uniform treatment of the propositional and first-order embeddings. Furthermore, we have discussed a new classification scheme in Section 1.3, which may provide better insights into the relation between different approaches, once they are classified.

8.3.4 The Relation to Neuro- and Cognitive Science

Even though artificial neural networks have been designed to mimic the brain, little biology is left in the architectures used today. Also in this thesis, we have not much been concerned with the relation to neuro- and cognitive science, and neither with biological plausibility. But as already mentioned in Section 8.2.6, recent neuroscience findings indicate that a symbolic understanding of the processes within biological neural networks might be feasible.

In [HK09], Pascal Hitzler and Kai-Uwe Kühnberger argue that the area of neural-symbolic integration is of central importance for the advancement of artificial general intelligence.

Also the recent work by Steffen Hölldobler and Carroline Dewi Puspa Kencana Ramli indicates that the relation between neural-symbolic systems and ideas from the cognitive science should be studied in details.

8.3.5 Symbolic-Neural Systems

Currently, neural-symbolic integration is about the embedding of logic into connectionist systems. I.e., about the connectionist implementation of symbolic processes. But we might also be able to carry over ideas from the connectionist to the symbolic side. One possible option is the application of gradient descent based learning techniques to annotated logic programs while using real numbers as annotations [KS92].

8.3.6 Direct Follow-Up Work wrt. This Thesis

The technical content presented in this thesis can be extended into various directions. Below we discuss some possible follow-up work structured according to the main chapters.

Embedding Propositional Rules into Connectionist Systems

In some application domains, we might encounter inconsistent symbolic background knowledge. E.g., if several sub-problems have been formalised by different people. If one of them is an expert and one of them is not, we should trust the expert more. In this case, it might be helpful to add confidence-values to the rules. Or even within a rule we might be very certain about some of the preconditions and not entirely sure about others. If we can annotate rules as well as preconditions of rules by confidence values, it should be very easy to integrate them into the propositional approach presented in Section 3. E.g., we could use an extended syntax for rules like,

$$0.3 : a \leftarrow 4 : b \wedge 0.3 : c$$

to express that we are very sure about the impact of b on a, not so sure about the impact of c, and neither very sure about the rule at all. The corresponding weights in the networks could simply be multiplied by those values to achieve the desired result. How this is related to [dGBG00] needs to be clarified.

One problem of the approach presented in Section 3 is the number of hidden units if many rules are available, because for each rule a new hidden unit is inserted. It might be feasible to pack the hidden layer, by distributing the rules over various units, but how this can be done is not clear yet.

Connectionist Learning and Propositional Background Knowledge

In Chapter 5, we discuss how to use symbolic knowledge together with connectionist learning algorithms. But the presented approach is rather naive in the rules used and we believe that much richer knowledge could also be used and integrated this way. How this can be achieved in detail needs to be worked out.

A completely different approach would be the "lifting" of connectionist learning paradigms to the symbolic side. Considering labelled logic programs, where atoms and rules are annotated with real values, we could apply a modified backpropagation directly on those values. We could furthermore use techniques from vector-based networks to split rules, or prune unimportant ones.

Extracting Propositional Rules from Connectionist Systems

All rule extraction algorithms have only been applied to rather small networks. This might be due to their complexity or due to the fact that huge rule-set are as incomprehensible as the set of weights itself. Nevertheless, those techniques could be used to gain insights into the internal structure of the network and how the acquired knowledge is distributed over the units. This could be done by extracting not only the output units, but the internal units of the network and study their symbolic meaning. This might also help to solve the above mentioned packing-problem for the hidden layer.

Another nice application for the rule extraction would be the analysis of the TD-Gammon player [Tes95]. This neural network based player for backgammon plays on a master level. During the training phase, the system shows a similar behaviour as novice human players show, in the sense that certain rules are learned very quickly and others only in later stages of the process. We might learn to better play the game by understanding its internal rules.

Existing extraction algorithms accept a neural network as input and produce a set of symbolic rules as output. The internal results are usually and probably on purpose hidden from the user. But we might even show (the rather simple) rules extracted for a hidden unit to an expert and ask her for the name of the corresponding concept. This higher-level name could then be used within the final rules. How this can be incorporated into existing extraction algorithms need to be explored, but an integration into the approach presented in Chapter 6 should be fairly easy. After extracting the set of minimal coalitions and oppositions for a unit, those sets could be shown to the expert.

Further details of the CoOp approach as presented in Chapter 6 need to be explored. In particular, the application of the approach to non-threshold units and detailed performance tests should be done.

Embedding First-Order Rules into Connectionist Systems

In Section 7 we showed how to embed first-order rules into connectionist systems. We presented constructions for sigmoidal and RBF networks for the 1-dimensional embedding. But there must also be constructions for the multi-dimensional case. This follows from Funahashi's theorem and the fact that even for the multi-dimensional case the embedding function is continuous and defined on a compact subset only. But what those constructions look like we do not know yet, and it still remains an open problem.

Another interesting foundation for the representation of first-order rules within connectionist model might be the *rational models* described in [Bor96]. They form a finite representation of infinite first-order interpretations. Unfortunately, their size is not bounded, even though it is finite. But there might be upper bounds which guarantee a sufficient approximation. If bounds can be established, it should be possible to represent those rational interpretations within a connectionist system.

Connectionist Learning and First-Order Background Knowledge

As mentioned above, the work described in Chapter 5 is only a first step for the usage of propositional knowledge during the training process. But the general idea presented there, should also be applicable within the first-order setting.

Extracting First-Order Rules from Connectionist Systems

The extraction of first-order rules from connectionist systems is probably the hardest problem in the field. There is not even a single approach how this could be achieved.

Nonetheless, Fibring networks as described in [BHdG05] might help to provide a starting point to solve the problem. Fibring allows for some kind of modularity in the sense that a network is composed of several sub-networks, which fibre each other, i.e., modify the weights. As shown in [BHdG05] and [KS05], those networks can be used to represent the semantic operator for first-order programs within a connectionist system. It should be possible to relate a symbolic meaning to those smaller parts of the system, and to the way they influence each other. Due to the modularity of the whole system, we can concentrate on those smaller parts, which should ease the problem. One could also use a set of prototype networks with a known meaning, which are then "just" composed using the fibring methodology.

8.4 Final Remarks

The underlying assumption of this thesis is that symbolic rules and connectionist systems can be conjoined within one system such that the strength of either paradigm is preserved. The methodology followed here is that of the neural-symbolic cycle as shown in Figure 1.1 on page 4. I.e., starting from some (symbolic) background knowledge, we construct an equivalent connectionist system which is then trained using standard learning techniques. The refined knowledge can afterwards be extracted to obtain a human readable form.

In this thesis, we discussed the embedding of symbolic rules into connectionist systems in great detail. We also discussed a new approach for the extraction of refined rules. Furthermore, we showed how connectionist training algorithms can be influenced with respect to symbolic knowledge. And we discussed an application of some ideas in the area of natural language processing, in particular to part of speech tagging.

The propositional setting, i.e., the case that background knowledge is represented using propositional rules, is fairly well understood, but things get more complex as soon as we move to the first-order setting. Here, many questions are still open, in particular, the problem of extracting first order rules from a trained network. The field of *neural-symbolic* integration remains an interesting research area. Not only because there are interesting problems that need to be solved, but also because it may lead to systems outperforming conventional approaches.

Bibliography

[AD99] Martin J. Adamson and Robert I. Damper. B-RAAM: A connectionist model which develops holistic internal representations of symbolic structures. *Connection Science*, 11(1):41–71, 1999. ↗p. 10

[ADT95] R. Andrews, J. Diederich, and A. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge–Based Systems*, 8(6), 1995. ↗p. 11, 18, 86

[AG99] Robert Andrews and Shlomo Geva. On the effect of initialising a neural network with prior knowledge. In *Proceedings of the International Conference on Neural Information Processing (ICONIP'99)*, pages 251–256, 1999. ↗p. 10, 11

[AN07] A. Asuncion and D. J. Newman. UCI machine learning repository, 2007. ↗p. 22, 79, 167

[And99] Henrik Reif Andersen. An introduction to binary decision diagrams. Lecture Notes, 1999. ↗p. 41, 43

[Bad03] Sebastian Bader. From logic programs to iterated function systems. Master's thesis, Department of Computer Science, Technische Universität Dresden, Dresden, Germany, May 2003. ↗p. 7, 133, 166

[Bad09] Sebastian Bader. Extracting propositional rules from feed-forward neural networks by means of binary decision diagrams. In Artur S. d'Avila Garcez and Pascal Hitzler, editors, *Proceedings of the 5th International Workshop on Neural-Symbolic Learning and Reasoning, NeSy'09, held at IJCAI-09*, pages 22 – 27, Pasadena, USA, JUL 2009. ↗p. 23, 85

[Bar93] Michael Barnsley. *Fractals Everywhere*. Academic Press, San Diego, CA, USA, 1993. ↗p. 30, 33, 34

[BDJ+99] Howard A. Blair, Fred Dushin, David W. Jakel, Angel J. Rivera, and Metin Sezgin. *The Logic Programming Paradigm: A 25-Year Persepective*, chapter Continuous models of computation for logic programs, pages 231–255. Springer, 1999. ↗p. 21, 166

[BDT05] Ernesto Burattini, Edoardo Datteri, and Guglielmo Tamburrini. Neuro-symbolic programs for robots. In *Proceedings of the IJCAI-05 workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK, August 2005*, 2005. ↗p. 168

[BG95] Enrico Blanzieri and Attilio Giordana. Mapping symbolic knowledge into locally receptive field networks. In *Topics in Artificial Intelligence, volume 992 of Lectures Notes in Artificial Intelligence*, pages 267–278. Springer-Verlag, 1995. ↗p. 10

[BH04] Sebastian Bader and Pascal Hitzler. Logic programs, iterated function systems, and recurrent radial basis function networks. *Journal of Applied Logic*, 2(3):273–300, 2004. Special Issue on Neural-Symbolic Systems. ↗p. 7, 15, 17, 124, 133, 166

[BH05] Sebastian Bader and Pascal Hitzler. Dimensions of neural-symbolic integration — a structured survey. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 1, pages 167–194. King's College Publications, JUL 2005. ↗p. 1, 12, 22

[BHdG05] Sebastian Bader, Pascal Hitzler, and Artur S. d'Avila Garcez. Computing first-order logic programs by fibring artificial neural networks. In I. Russell and Z. Markov, editors, *Proceedings of the 18th International Florida Artificial Intelligence Research Symposium Conference, FLAIRS05, Clearwater Beach, Florida, May 2005*, pages 314–319. AAAI Press, 2005. ↗p. 7, 15, 17, 124, 172

[BHH04] Sebastian Bader, Pascal Hitzler, and Steffen Hölldobler. The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. In L. Li and K.K. Yen, editors, *Proceedings of the Third International Conference on Information, Tokyo, Japan, November/December 2004*, pages 22–33. International Information Institute, Hosei University, Tokyo, Japan, 2004. ↗p. 1

[BHH06] Sebastian Bader, Pascal Hitzler, and Steffen Hölldobler. The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. *Journal of Information*, 9(1):7–20, January 2006. ↗p. 18, 22

[BHH08] Sebastian Bader, Pascal Hitzler, and Steffen Hölldobler. Connectionist model generation: A first-order approach. *Neurocomputing*, 71(13–15):2420–2432, AUG 2008. ↗p. 7, 16, 23

[BHHW07] Sebastian Bader, Pascal Hitzler, Steffen Hölldobler, and Andreas Witzel. A fully connectionist model generator for covered first-order logic programs. In Manuela M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India*, pages 666–671, Menlo Park CA, January 2007. AAAI Press. ↗p. 7, 23, 123, 124, 150, 157

[BHM08] Sebastian Bader, Steffen Hölldobler, and Nuno C. Marques. Guiding backprop by inserting rules. In Artur S. d'Avila Garcez and Pascal Hitzler, editors, *Proceedings of the 4th International Workshop on Neural-Symbolic Learning and Reasoning, NeSy'08, held at ECAI-08*, volume 366 of *CEUR Workshop Proceedings*, JUL 2008. ↗p. 20, 22, 77

[BHME07] Sebastian Bader, Steffen Hölldobler, and Valentin Mayer-Eichberger. Extracting propositional rules from feed-forward neural networks — a new decompositional approach. In Artur S. d'Avila Garcez and Pascal Hitzler, editors, *Proceedings of the 3rd International Workshop on Neural-Symbolic Learning and Reasoning, NeSy'07, held at IJCAI-07*, JAN 2007. ↗p. 23, 85

[BHS04] Sebastian Bader, Steffen Hölldobler, and Alexandre Scalzitti. Semiring artificial neural networks and weighted automata. In G. Palm S. Biundo, T. Frühwirth, editor, *KI 2004: Advances in Artificial Intelligence. Proceedings of the 27th Annual*

German Conference on Artificial Intelligence, Ulm, Germany, September 2004, volume 3238 of *Lecture Notes in Artificial Intelligence*, pages 281–294. Springer, 2004. ↗p. 4, 16

[BHW05] Sebastian Bader, Pascal Hitzler, and Andreas Witzel. Integrating first-order logic programs and connectionist systems — a constructive approach. In Artur S. d'Avila Garcez, Jeff Elman, and Pascal Hitzler, editors, *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, held at IJCAI-05*, Edinburgh, UK, 2005. ↗p. 7, 17, 23, 123, 124

[Bib87] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg Verlag, Braunschweig, 2nd edition, 1987. ↗p. 11

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995. ↗p. 9, 45

[Bla97] A. Blair. Scaling-up RAAMs. Technical Report CS-97-192, University of Brandeis, 1997. ↗p. 8

[Bol00] Guido Bologna. Rule extraction from a multi layer perceptron with staircase activation functions. In *IJCNN (3)*, pages 419–424, 2000. ↗p. 18

[Bor96] Sven-Erik Bornscheuer. Rational models of normal logic programs. In *KI-96: Advances in Artificial Intelligence*, number 1137 in LNAI, pages 1–4. Springer, 1996. ↗p. 17, 124, 171

[BS99] Anthony Browne and Ron Sun. Connectionist variable binding. *Expert Systems*, 16(3):189–207, 1999. ↗p. 17

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996. ↗p. 45

[BZB01] Rodrigo Basilio, Gerson Zaverucha, and Valmir C. Barbosa. Learning logic programs with neural networks. In *Proceedings of the 11th International Conference on Inductive Logic Programming*, pages 15–26, London, UK, 2001. Springer-Verlag. ↗p. 10

[DAG98] WDuch, RafałAdamczak, and Krzysztof Grabczewski. Extraction of logical rules from neural networks. *Neural Processing Letters*, 7(3):211–219, 1998. ↗p. 11

[DAG01] WDuch, RafałAdamczak, and Krzysztof Grabczewski. A new methodology of extraction, optimization and application ofcrisp and fuzzy logical rules. *IEEE Transactions on Neural Networks*, 12(2):277–306, Mar 2001. ↗p. 11

[Dar00] Ashish Darbari. First-order rule learning through a pulsed neural network. Master's thesis, Technische Universität Dresden, 2000. ↗p. 14

[dCNFR04] M. do Canno Nicoletti, L.B. Figueira, and A. Ramer. Transferring symbolic knowledge into a rulenet neural network. In *IEEE International Conference on Computational Cybernetics*, pages 317 – 322, Vienna, Austria, SEP 2004. ↗p. 11

[dGBG00] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. Metalevel priorities and neural networks. In *Proceedings of the Workshop on the Foundations of Connectionist-Symbolic Integration, ECAI'2000, Berlin*, August 2000. ↗p. 6, 17, 170

[dGBG01] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125:155–207, 2001. ↗p. 12, 18, 19

[dGBG02] Artur S. d'Avila Garcez, Krysia B. Broda, and Dov M. Gabbay. *Neural-Symbolic Learning Systems — Foundations and Applications*. Perspectives in Neural Computing. Springer, Berlin, 2002. ↗p. 22

[dGG04] Artur S. d'Avila Garcez and Dov M. Gabbay. Fibring neural networks. In *In Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 04). San Jose, California, USA, July 2004*. AAAI Press, 2004. ↗p. 7, 17

[dGGL04] Artur S. d'Avila Garcez, Dov M. Gabbay, and Luis C. Lamb. Argumentation neural networks. In *Proceedings of 11th International Conference on Neural Information Processing (ICONIP'04)*, Lecture Notes in Computer Science LNCS, Calcutta, November 2004. Springer-Verlag. ↗p. 17

[dGGL05a] Artur S. d'Avila Garcez, Dov M. Gabbay, and Luis C. Lamb. *Connectionist Non-Classical Logics*. Springer-Verlag, 2005. To appear. ↗p. 6, 17, 169

[dGGL05b] Artur S. d'Avila Garcez, Dov M. Gabbay, and Luis C. Lamb. Value-based argumentation frameworks as neural-symbolic learning systems. *Journal of Logic and Computation*, 2005. To appear. ↗p. 17

[dGLBG04] Artur S. d'Avila Garcez, Luis C. Lamb, Kryisa Broda, and Dov M. Gabbay. Applying connectionist modal logics to distributed knowledge representation problems. *International Journal of Artificial Intelligence Tools*, 2004. ↗p. 17

[dGLG02] Artur S. d'Avila Garcez, Luis C. Lamb, and Dov M. Gabbay. A connectionist inductive learning system for modal logic programming. In *Proceedings of the IEEE International Conference on Neural Information Processing ICONIP'02, Singapore*, 2002. ↗p. 6, 17

[dGLG03] Artur S. d'Avila Garcez, Luis C. Lamb, and Dov M. Gabbay. Neural-symbolic intuitionistic reasoning. In M. Koppen A. Abraham and K. Franke, editors, *Frontiers in Artificial Intelligence and Applications*, Melbourne, Australia, December 2003. IOS Press. Proceedings of the Third International Conference on Hybrid Intelligent Systems (HIS'03). ↗p. 6, 17

[dGZ99] Artur S. d'Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence, Special Issue on Neural networks and Structured Knowledge*, 11(1):59–77, 1999. ↗p. 6, 15, 18, 58, 67, 162

[dGZdC97] Artur S. d'Avila Garcez, Gerson Zaverucha, and Luis A. V. de Carvalho. Logical inference and inductive learning in artificial neural networks. In Christoph Hermann, Frank Reine, and Antje Strohmaier, editors, *Knowledge Representation in Neural networks*, pages 33–46. Logos Verlag, Berlin, 1997. ↗p. 6, 18, 22

[Elm90] Jeff L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990. ↗p. 16, 167

[FCBGM99] Mikel L. Forcada, Antonio M. Corbí-Bellot, Marco Gori, and Marco Maggini. Neural learning of approximate simple regular languages. In *Proceedings of ESANN'99*, pages 57–62, 1999. ↗p. 16

[FGKS01] P. Frasconi, M. Gori, A. Kuchler, and A. Sperduti. From sequences to data structures: Theory and applications. In J. Kolen and S.C. Kremer, editors, *Dynamical Recurrent Networks*, pages 351–374. IEEE Press, 2001. ↗p. 10, 16

[Fle01] Peter Fletcher. Connectionist learning of regular graph grammars. *Connection Science*, 13(2):127–188, 2001. ↗p. 16

[Fri98] Bernd Fritzke. *Vektorbasierte Neuronale Netze.* Habilitation, Technische Universität Dresden, 1998. ↗p. 51, 157, 158, 159

[Fun89] K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989. ↗p. 50, 124

[Gab99] Dov M. Gabbay. *Fibring Logics.* Oxford Univesity Press, 1999. ↗p. 17

[GCM+91] C. Giles, D. Chen, C. Miller, H. Chen, G. Sun, and Y. Lee. Second-order recurrent neural networks for grammatical inference. In *Proceedings of the International Joint Conference on Neural Networks 1991*, volume 2, pages 273–281, New York, 1991. IEEE. ↗p. 16, 18

[GK05] Helmar Gust and Kai-Uwe Kühnberger. Learning symbolic inferences with neural networks. In B. Bara, L. Barsalou, and M. Bucciarelli, editors, *CogSci 2005: XXVII Annual Conference of the Cognitive Science Society*, pages 875–880, 2005. ↗p. 11, 15, 17, 169

[Ham98] J.A. Hammerton. *Exploiting Holistic Computation: An evaluation of the Sequential RAAM.* PhD thesis, University of Birmingham, 1998. ↗p. 10

[Ham02] Barbara Hammer. Recurrent networks for structured data — a unifying approach and its properties. *Cognitive Systems Research*, 3(2):145–165, 2002. ↗p. 16

[Ham03] Barbara Hammer. Perspectives on learning symbolic data with connectionistic systems. In R. Kühn, R. Menzel, W. Menzel, U. Ratsch, M.M. Richter, and I.-O. Stamatescu, editors, *Adaptivity and Learning*, pages 141–160. Springer, 2003. ↗p. 16

[Han01] Leitgeb Hannes. Nonmonotonic reasoning by inhibition nets. *Artificial Intelligence*, 128(1):161–201(41), May 2001. ↗p. 17

[HBdG05] Pascal Hitzler, Sebastian Bader, and Artur S. d'Avila Garcez. Ontology learning as a use-case for neural-symbolic integration — position paper. In *Proceedings of the IJCAI-05 workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK, August 2005*, 2005. ↗p. 168

[Heb49] D. O. Hebb. *The Organization of Behavior.* Wiley, New York, 1949. ↗p. 15

[HHS04] Pascal Hitzler, Steffen Hölldobler, and Anthony K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 3(2):245–272, 2004. ↗p. 7, 17, 58, 66

[Hil95] Melanie Hilario. An overview of strategies for neurosymbolic integration. In R. Sun and F. Alexandre, editors, *Proceedings of the Workshop on Connectionist-Symbolic Integration: From Unied to Hybrid Approaches*, Montreal, 1995. ↗p. 12

[Hit04] Pascal Hitzler. Corollaries on the fixpoint completion: studying the stable semantics by means of the clark completion. In D. Seipel, M. Hanus, U. Geske, and O. Bartenstein, editors, *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management and the 18th Workshop on Logic Programming, Potsdam, Germany, March 4-6, 2004*, volume 327 of *Technichal Report*, pages 13–27. Bayerische Julius-Maximilians-Universität Würzburg, Institut für Informatik, 2004. ↗p. 17

[HK92] S. Hölldobler and F. Kurfess. CHCL – A connectionist inference system. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, pages 318 – 342. Springer, LNAI *590*, 1992. ↗p. 11, 16

[HK94] Steffen Hölldobler and Yvonne Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994. ↗p. 4, 6, 7, 16, 17, 22, 55, 58, 67, 162

[HK09] Pascal Hitzler and Kai-Uwe Kühnberger. The importance of being neural-symbolic - a wilde position. In Ben Goertzel, Pascal Hitzler, and Marcus Hutter, editors, *Artificial General Intelligence. Second Conference on Artificial General Intelligence, AGI 2009*, Arlington, Virginia, USA, March 2009. ↗p. 170

[HKL97] Steffen Hölldobler, Yvonne Kalinke, and Helko Lehmann. Designing a counter: Another case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the KI97: Advances in Artificial Intelligence*, volume 1303 of *LNAI*, pages 313–324. Springer Berlin / Heidelberg, 1997. ↗p. 16

[HKS99] Steffen Hölldobler, Yvonne Kalinke, and Hans-Peter Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999. ↗p. 7, 17, 123, 124, 134, 144, 154, 157, 159, 163

[HKW99] Steffen Hölldobler, Yvonne Kalinke, and Jörg Wunderlich. A recursive neural network for reflexive reasoning. In Stefan Wermter and Ron Sun, editors, *Hybrid Neural Systems*. Springer, Berlin, 1999. ↗p. 10, 17

[HMSS04a] Barbara Hammer, A. Micheli, A. Sperduti, and M. Strickert. Recursive self-organizing network models. *Neural Networks*, 17(8–9):1061–1085, 2004. Special issue on New Developments in Self-Organizing Systems. ↗p. 16

[HMSS04b] Barbara Hammer, A. Micheli, M. Strickert, and A. Sperduti. A general framework for unsupervised processing of structured data. *Neurocomputing*, 57:3–35, 2004. ↗p. 16

[Höl90] Steffen Hölldobler. A structured connectionist unification algorithm. In *Proceedings of AAAI*, pages 587–593, 1990. ↗p. 11, 16

[Höl93] Steffen Hölldobler. *Automated Inferencing and Connectionist Models*. Fakultät Informatik, Technische Hochschule Darmstadt, 1993. Habilitationsschrift. ↗p. 16

[Höl00] Steffen Hölldobler. Challenge problems for the integration of logic and connectionist systems. In F. Bry, U. Geske, and D. Seipel, editors, *Proceedings 14. Workshop Logische Programmierung*, pages 161–170, 2000. ↗p. 18

[HP04] I. Hatzilygeroudis and J. Prentzas. Neuro-symbolic approaches for knowledge representation in expert systems. *International Journal of Hybrid Intelligent Systems*, 1(3-4):111–126, 2004. ↗p. 12

[HR09a] Steffen Hölldobler and Carroline Dewi Puspa Kencana Ramli. Logic programs under three-valued sementics. In *Proceedings of the 25th International Conference on Logic Programming (ICLP)*, Pasadena, USA, JUL 2009. to appear. ↗p. 169

[HR09b] Steffen Hölldobler and Carroline Dewi Puspa Kencana Ramli. Logics and networks for human reasoning. In *Proceedings of the 19th International Conference on Artificial Neural Networks (ICANN)*, Limassol, Cyprus, September 2009. to appear. ↗p. 169

[HS00] Pascal Hitzler and Anthony K. Seda. A note on relationships between logic programs and neural networks. In Paul Gibson and David Sinclair, editors, *Proceedings of the Fourth Irish Workshop on Formal Methods, IWFM'00*, Electronic Workshops in Comupting (eWiC). British Computer Society, 2000. ↗p. 7, 17

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989. ↗p. 124

[HTSK08] Markus Hagenbuchner, Ah Chung Tsoi, Alessandro Sperduti, and Milly Kc. Efficient clustering of structured documents using graph self-organizing maps. pages 207–221, 2008. ↗p. 16

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979. ↗p. 5

[Jac05] H. Jacobsson. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Computation*, 17(6):1223–1263, 2005. ↗p. 11, 18, 86

[JBN04] R. P. Jagadeesh, Chandra Bose, and G. Nagaraja. Performance studies on kbann. In *Fourth International Conference on Hybrid Intelligent Systems, HIS '04*, pages 198–203, Washington, DC, USA, DEC 2004. IEEE Computer Society. ↗p. 11, 83

[Kal94] Yvonne Kalinke. Ein massiv paralleles Berechnungsmodell für normale logische Programme. Master's thesis, TU Dresden, Fakultät Informatik, 1994. (in German). ↗p. 6

[Kal97] Yvonne Kalinke. Using connectionist term representation for first–order deduction – a critical view. In F. Maire, R. Hayward, and J. Diederich, editors, *Connectionist Systems for Knowledge Representation Deduction*. Queensland University of Technology, 1997. CADE–14 Workshop, Townsville, Australia. ↗p. 8

[KK95] S. Kwasny and B. Kalman. Tail-recursive distributed representations and simple recurrent networks. *Connection Science*, 7:61–80, 1995. ↗p. 10

[KL98] Yvonne Kalinke and Helko Lehmann. Computations in recurrent neural networks: From counters to iterated function systems. In G. Antoniou and J. Slaney, editors, *Advanced Topics in Artificial Intelligence*, volume 1502 of *LNAI*, Berlin/Heidelberg, 1998. Proceedings of the 11th Australian Joint Conference on Artificial Intelligence (AI'98), Springer–Verlag. ↗p. 16

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956. ↗p. 4, 16

[KS92] Michael Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *The Journal of Logic Programming*, 12(4):335–367, April 1992. ↗p. 170

[KS05] Vladimir Komendantsky and Anthony Seda. Computation of normal logic programs by fibring neural networks. In *Proceedings of the Seventh International Workshop on First-Order Theorem Proving (FTP'05)*, pages 97–111, Koblenz, Germany, SEP 2005. ↗p. 7, 172

[KSB99] Ramanathan Krishnan, G. Sivakumar, and Pushbak Bhattacharya. A search technique for rule extraction from trained neural networks. *Non-Linear Anal.*, 20(3):273–280, 1999. ↗p. 12, 19

[LBH05] Jens Lehmann, Sebastian Bader, and Pascal Hitzler. Extracting reduced logic programs from artificial neural networks. In Artur S. d'Avila Garcez, Jeff Elman, and Pascal Hitzler, editors, *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK*, 2005. ↗p. 18

[LDS+90] Y. LeCun, J. Denker, S. Solla, R. E. Howard, and L. D. Jackel. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems II*, San Mateo, CA, 1990. Morgan Kauffman. ↗p. 83

[Llo87] John Wylie Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 2nd extended edition, 1987. ↗p. 34

[LS06] Máire Lane and Anthony K. Seda. Some aspects of the integration of connectionist and logic-based systems. *Information*, 9(4):551–562, 2006. ↗p. 7

[Maa02] Wolfgang Maass. Paradigms for computing with spiking neurons. In J. L. van Hemmen, J. D. Cowan, , and E. Domany, editors, *Models of Neural Networks*, volume 4 of *Early Vision and Attention*, chapter 9, pages 373–402. Springer, 2002. ↗p. 14

[MBRH07] Nuno C. Marques, Sebastian Bader, Vitor Rocio, and Steffen Hölldobler. Neuro-symbolic word tagging. In José Machado José Neves, Manuel Filipe Santos, editor, *New Trends in Artificial Intelligence*, pages 779–790. APPIA - Associação Portuguesa para a Inteligência Artificial, 12 2007. ↗p. 69, 166, 167

[Mit97] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, March 1997. ↗p. 80

[ML01] Nuno C. Marques and Gabriel Pereira Lopes. Neural networks, part-of-speech tagging and lexicons. In *Proceedings of the International Conference on Intelligent Data Analysis (IDA'01)*, number 2189 in LNCS, pages 63–72, Cascais, Portugal, September 2001. Springer Verlag. ↗p. 70

[MM04] W. Maass and H. Markram. On the computational power of recurrent circuits of spiking neurons. *Journal of Computer and System Sciences*, 69(4):593–616, 2004. ↗p. 14

[MNM05] W. Maass, T. Natschläger, and H. Markram. On the computational power of circuits of spiking neurons. *J. of Physiology (Paris)*, 2005. in press. ↗p. 14

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943. ↗p. 4, 14, 16

[MTWM99] K. J. McGarry, J. Tait, S. Wermter, and J. MacIntyre. Rule-extraction from radial basis function networks. In *Ninth International Conference on Artificial Neural Networks (ICANN'99)*, volume 2, pages 613–618, Edinburgh, UK, 1999. ↗p. 11, 18

[NM02] T. Natschläger and W. Maass. Spiking neurons and the induction of finite state machines. *Theoretical Computer Science: Special Issue on Natural Computing*, 287:251–265, 2002. ↗p. 14

[Pin91a] G. Pinkas. Propositional non-monotonic reasoning and inconsistency in symmetrical neural networks. In *IJCAI*, pages 525–530, 1991. ↗p. 17

[Pin91b] G. Pinkas. Symmetric neural networks and logic satisfiability. *Neural Computation*, 3:282–291, 1991. ↗p. 10, 17, 18

[Pin94] Gadi Pinkas. *Artificial Intelligence and Cognitive Modeling: Steps towards Principled Integration*, chapter A fault tolerant connectionist architecture for construction of logic proofs. Academic Press, Academic Press, 1994. ↗p. 11

[Pla91] Tony A. Plate. Holographic Reduced Representations: Convolution algebra for compositional distributed representations. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence, Sydney, Australia, August 1991*, pages 30–35, San Mateo, CA, 1991. Morgan Kauffman. ↗p. 10

[Pla95] Tony A. Plate. Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6(3):623–641, May 1995. ↗p. 10

[Pol88] Jordan B. Pollack. Recursive auto-associative memory: Devising compositional distributed representations. In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 33–39, 1988. ↗p. 8, 14, 15

[Pol90] Jordan B. Pollack. Recursive distributed representations. *AIJ*, 46:77–105, 1990. ↗p. 8, 9, 16

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart, J.L. McClelland, and the PDP Research Group, editors, *Parallel Distributed Processing, vol. 1: Foundations*, pages 318–362. MIT Press, 1986. ↗p. 15

[Roj96] Raul Rojas. *Neural Networks*. Springer, 1996. ↗p. 45

[SA93] Lokendra Shastri and Venkat Ajjanagadde. From associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioural and Brain Sciences*, 16(3):417–494, September 1993. ↗p. 10, 14, 15

[Sam95] Geoffrey Sampson. *English for the Computer: The SUSANNE Corpus and Analytic Scheme*. Oxford University Press, Oxford, 1995. ↗p. 70

[Sch94] H. Schmid. Part-of-speech tagging with neural networks. In *Proceeding of COLING-94*, pages 172–176, 1994. ↗p. 70, 167

[Sed05] Anthony K. Seda. On the integration of connectionist and logic-based systems. In T. Hurley, M. Mac an Airchinnigh, M. Schellekens, A.K. Seda, and G. Strong, editors, *Proceedings of MFCSIT2004, Trinity College Dublin, July 2004*, Electronic Notes in Theoretical Computer Science, pages 1–24. Elsevier, 2005. ↗p. 7, 17

[SGT+08] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 2008. To appear. ↗p. 16

[Sha99] Lokendra Shastri. Advances in Shruti — A neurally motivated model of relational knowledge representation and rapid inference using temporal synchrony. *Applied Intelligence*, 11:78–108, 1999. ↗p. 10

[Sha02] Lokendra Shastri. Episodic memory and cortico-hippocampal interactions. *Trends in Cognitive Sciences*, 6:162–168, 2002. ↗p. 15

[SL05] Anthony K. Seda and Máire Lane. On approximation in the integration of connectionist and logic-based systems. In *Proceedings of the Third International Conference on Information (Information'04)*, pages 297–300, Tokyo, November 2005. International Information Institute. ↗p. 7, 15, 17

[Sou01] Jacques P. Sougne. Binding and multiple instantiation in a distributed network of spiking nodes. *Connection Science*, 13(1):99–126, 2001. ↗p. 14

[Spe94a] A. Sperduti. Labeling RAAM. *Connection Science*, 6(4):429–459, 1994. ↗p. 8, 169

[Spe94b] Alessandro Sperduti. Encoding labeled graphs by labeling RAAM. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, pages 1125–1132. Morgan Kaufmann, 1994. ↗p. 8

[SSG95] Alessandro Sperduti, Antonina Starita, and Christoph Goller. Learning distributed representations for the classifications of terms. In *Proceedings of the 14th International Joint Conference on AI, IIJCAI-95*, pages 509–517. Morgan Kaufmann, 1995. ↗p. 10, 16

[SSG97] Alessandro Sperduti, Antonina Starita, and Christoph Goller. Distributed representations for terms in hybrid reasoning systems. In Ron Sun and Frédéric Alexandre, editors, *Connectionist Symbolic Integration*, chapter 18, pages 329–344. Lawrence Erlbaum Associates, 1997. ↗p. 10, 16

[Sun01] Ron Sun. Hybrid systems and connectionist implementationalism. In *Encyclopedia of Cognitive Science*. MacMillan Publishing Company, 2001. ↗p. 14

[SvL08] Keith Stenning and Michiel van Lambalgen. *Human Reasoning and Cognitive Science*. MIT Press, AUG 2008. ↗p. 169

[SW92] Andreas Stolcke and Dekai Wu. Tree matching with recursive distributed representations. Technical Report tr-92-025, ICSI, Berkeley, 1992. ↗p. 8

[SW99] Lokendra Shastri and Carter Wendelken. Soft computing in SHRUTI: — A neurally plausible model of reflexive reasoning and relational information processing. In *Proceedings of the Third International Symposium on Soft Computing, Genova, Italy*, pages 741–747, June 1999. ↗p. 10

[SW03] Lokendra Shastri and C. Wendelken. Learning structured representations. *Neurocomputing*, 52–54:363–370, 2003. ↗p. 10, 15

[Tes95] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), March 1995. ↗p. 171

[TS93] Geoffrey G. Towell and Jude W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, 1993. ↗p. 18, 19

[TS94] Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1–2):119–165, 1994. ↗p. 7, 8, 11, 15, 22, 67, 166

[Val03] Leslie G. Valiant. Three problems in computer science. *Journal of the ACM*, 50(1):96–99, 2003. ↗p. 18

[vdVdK05] Frank van der Velde and Marc de Kamps. Neural blackboard architectures of combinatorial structures in cognition. *Behavioral and Brain Sciences*, 2005. to appear. ↗p. 14, 16

[vH99] Hans van Halteren. *Syntactic Wordclass Tagging*. Kluwer Academic Publishers, 1999. ↗p. 70, 167

[Wer74] Paul J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*. PhD thesis, Harvard University, Cambridge, Mass., November 1974. Reprinted in [Wer94]. ↗p. 15

[Wer94] Paul J. Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley, New York, NY, 1994. ↗p. 183

[Wie09] Jan Wielemaker. *Logic programming for knowledge-intensive interactive applications*. PhD thesis, Faculty of Science, Universiteit van Amsterdam, JUN 2009. ↗p. 26

[Wil70] S. Willard. *General Topology*. Addison–Wesley, 1970. ↗p. 129

[Wit05] Andreas Witzel. Integrating first-order logic programs and connectionist systems - a constructive approach. Project thesis, Department of Computer Science, Technische Universität Dresden, Dresden, Germany, 2005. ↗p. 7

[Wit06] Andreas Witzel. Neural-symbolic integration – constructive approaches. Master's thesis, Department of Computer Science, Technische Universität Dresden, Dresden, Germany, 2006. ↗p. 150

[WS03] C. Wendelken and Lokendra Shastri. Acquisition of concepts and causal rules in shruti. In *Proceedings of Cognitive Science, Boston, MA*, August 2003. ↗p. 10, 15

[WS04] C. Wendelken and Lokendra Shastri. Multiple instantiation and rule mediation in shruti. *Connection Science*, 16:211–217, 2004. ↗p. 10

[www09a] Buddy: http://buddy.sourceforge.net, JUL 2009. ↗p. 41

[www09b] CUDD: http://vlsi.colorado.edu/~fabio/CUDD, JUL 2009. ↗p. 41

[www09c] SWI-prolog: http://www.swi-prolog.org, JUN 2009. ↗p. 26

[YS07] Tianming Yang and Michael N. Shadlen. Probabilistic reasoning by neurons. *Nature*, 447(7148):1075–1080, 06 2007. ↗p. 166

[Zel92] Andreas Zell. SNNS, stuttgart neural network simulator, user manual, version 2.1. Technical report, Stuttgart, 1992. ↗p. 74, 80