Connectionist Model Generation: A First-Order Approach

Sebastian Bader *[1], Pascal Hitzler[2], Steffen Hölldobler[1]

[1] *International Center for Computational Logic, Technische Universität Dresden, 01062 Dresden, Germany*

[2] *Institute AIFB, Universität Karlsruhe (TH), 76128 Karlsruhe, Germany*

Abstract

Knowledge based artificial neural networks have been applied quite successfully to propositional knowledge representation and reasoning tasks. However, as soon as these tasks are extended to structured objects and structure-sensitive processes as expressed e.g., by means of first-order predicate logic, it is not obvious at all what neural symbolic systems would look like such that they are truly connectionist, are able to learn, and allow for a declarative reading and logical reasoning at the same time. The core method aims at such an integration. It is a method for connectionist model generation using recurrent networks with feed-forward core. We show in this paper how the core method can be used to learn first-order logic programs in a connectionist fashion, such that the trained network is able to do reasoning over the acquired knowledge. We also report on experimental evaluations which show the feasibility of our approach.

Key words: Connectionist Model Generation, Neural-Symbolic Integration, Recurrent RBF Networks, First-Order Logic Programs

1. Introduction

The integration of neural networks with systems based on symbolic knowledge representation and reasoning is a formidable challenge. It is motivated by the quest for a tight integration of the robustness and learning abilities of neural networks on the one hand, and the declarative nature and reasoning capabilities of symbolic systems on the

* Corresponding author.

Email addresses: `Sebastian.Bader@inf.tu-dresden.de` (Sebastian Bader), `hitzler@aifb.uni-karlsruhe.de` (Pascal Hitzler), `sh@iccl.tu-dresden.de` (Steffen Hölldobler).

other hand. Attempts towards such tight integrations seek for ways to represent logical knowledge in a connectionist setting, for ways of training neural networks with such knowledge, and to reason with the symbolic knowledge encoded in such a network.

From the very beginning, research focused on relations between artificial neural networks and propositional logic. McCulloch-Pitts networks, for example, can be viewed as computing propositional logical formulae [29]. Finding global minima of the energy function associated with a symmetric network corresponds to finding models of a propositional logic formula [30]. These are just two examples that illustrate what McCarthy has called a *propositional fixation* of connectionist systems in [28]. From the perspective of symbolic systems, however, capabilities beyond propositional logic are required.

There exists a reasonable body of research results on modeling first-order fragments in connectionist systems. In [7], energy minimization is used to model inference processes involving unary relations. In [26] and [33] multi-place predicates and rules over such predicates are modeled. In [23], a connectionist inference system for a limited class of logic programs was developed. But a deeper analysis of these and other systems reveals that the systems are in fact propositional, and furthermore, these systems usually either have good reasoning capabilities or can be trained, but are not able to both learn *and* reason in a powerful way. We are unaware of any connectionist system that fully incorporates learning and reasoning for structured objects in the sense of logic-based knowledge representation, and thus incorporates the power of symbolic computation as argued for in [34].

Here we are concerned with knowledge based artificial neural networks, i.e., with networks which are initialized by available background knowledge before training methods are applied. In [35] it has been shown that such networks perform better than purely empirical and hand-built classifiers. [35] used background knowledge in the form of propositional rules and encodes these rules in multi-layer feed-forward networks. Independently, we have developed along similar lines a connectionist system for computing the least model of (definite) propositional logic programs [21]. This system has been further developed into the so-called *core method*: background knowledge represented as logic programs is encoded in a multi-layer feed-forward network, training methods can be applied to the feed-forward kernel in order to improve the performance of the network, and recurrent connections allow for a computation or approximation of models of the logic program and thus for reasoning with the encoded knowledge. Finally, an improved program can be extracted from the trained core, which closes the neural-symbolic cycle.

We present the core method for first-order logic programs. After a general introduction in Section 3, we present in detail in Section 4 how to represent first-order logic programs within radial basis function (RBF) networks. The representation is based on the Cantor space as a bridge between the discrete paradigm of logic programming and the continuous paradigm of neural networks. In Section 5, we present our training algorithm and in Section 6 we report on an evaluation of our implementation, which shows that our system is indeed able to learn logic programs, is robust against unit failure, and can be used for reasoning in the sense that it is possible to compute the desired model of the target logic program. To the best of our knowledge, this is the first system which combines such capabilities for first-order logic programming within a purely connectionist architecture. These main sections are framed by introducing technical preliminaries in Section 2, discussions of further and related work in Sections 7 and 8, and a concluding Section 9. This paper is a substantially revised and extended version of [5] and [6] by including all necessary proofs, providing more intuition and more details of the training.

2. Technical Preliminaries

We assume the reader to be familiar with basic notions from artificial neural networks and logic programs and refer to [9] and [27], respectively. Nevertheless, we repeat some basic notions, which will be required in later sections. We will start with the notion of a metric space as it is crucial throughout the paper.

Definition 1 *Let X be a set and let $d : X \times X \to \mathbb{R}$ be a function. The function d is called a* metric *and (X, d) is called a* metric space *if for all $x, y, z \in X$ the following conditions hold: (i) $d(x, x) = 0$, (ii) $d(x, y) = d(y, x)$ and (iii) $d(x, z) \leq d(x, y) + d(y, z)$.*

For example, the following well known *maximum metric* over the m-dimensional real numbers will be used below:

$$d_{\mathbb{R}} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R} : (\boldsymbol{x}, \boldsymbol{y}) \mapsto \max_{1 \leq i \leq m} |x_i - y_i|.$$

The notion of continuity, known from functions on $\mathbb{R}$, can be generalized to metric spaces as follows. For two metric spaces (X_1, d_1) and (X_2, d_2), a function $f : X_1 \to X_2$ is called *continuous* if for all $\varepsilon > 0$ there exists $\delta > 0$ such that for all $x, y \in X_1$ with $d_1(x, y) < \delta$ we have $d(f(x), f(x)) < \varepsilon$.

2.1. *Logic Programs*

A *logic program* over some language $\mathcal{L}$ is a set of *clauses* of the form $A \leftarrow L_1 \wedge \cdots \wedge L_n$, where A is an *atom* in $\mathcal{L}$ and the L_i, $1 \leq i \leq n$, are *literals* in $\mathcal{L}$, that is, atoms or negated atoms. A is called the *head* of the clause, the L_i are called *body literals*, and their conjunction $L_1 \wedge \cdots \wedge L_n$ is called the *body* of the clause. If $n = 0$, A is called a *fact*. A clause is *ground* if it does not contain any variables. *Local variables* are those variables occurring in some body but not in the corresponding head. A logic program is *covered* if none of the clauses contain local variables. A logic program is *propositional* if all predicate letters occurring in the program are nullary.

Example 2 The following propositional logic program will serve as example in Section 3.

$$\begin{array}{lll}
\mathcal{P}_1 = \{ & p, & \textit{\% p is always true.} \\
& r \leftarrow p \wedge \neg q, & \textit{\% r is true if p is true and q is false.} \\
& r \leftarrow \neg p \wedge q \ \} & \textit{\% r is true if p is false and q is true.} \quad \star
\end{array}$$

Example 3 The following (first-order) logic program will serve as example in Section 4.

$$\begin{array}{lll}
\mathcal{P}_2 = \{ & e(0). & \textit{\% 0 is even} \\
& e(s(X)) \leftarrow o(X). & \textit{\% the successor s(X) of an odd X is even} \\
& o(X) \leftarrow \neg e(X). \quad \} & \textit{\% X is odd if it is not even} \quad \star
\end{array}$$

The *Herbrand universe* $\mathcal{U}_{\mathcal{L}}$ is the set of all ground terms of $\mathcal{L}$, the *Herbrand base* $\mathcal{B}_{\mathcal{L}}$ is the set of all ground atoms, which we assume to be infinite – indeed the case of a finite $\mathcal{B}_{\mathcal{L}}$ can be reduced to a propositional setting. A *ground instance* of a literal or a clause is obtained by replacing all variables by terms from $\mathcal{U}_{\mathcal{L}}$. For a logic program P, $\mathcal{G}(P)$ denotes the set of all ground instances of clauses from P.

A *Herbrand interpretation* I is a subset of $\mathcal{B}_{\mathcal{L}}$. Atoms $A \in I$ are said to be *true* under I, those $A \notin I$ are said to be *false* under I. $\mathcal{I}_{\mathcal{L}}$ denotes the set of all interpretations. An interpretation I is a *(Herbrand) model* of a logic program P (in symbols: $I \models P$) if I is a model for each clause in $\mathcal{G}(P)$ in the usual sense.

Example 4 For $\mathcal{P}_2$ we have $M_2 := \{e(s^n(0)) \mid n \text{ even}\} \cup \{o(s^m(0)) \mid m \text{ odd}\} \models P$. ⋆

Given a logic program P, the *single-step operator* or *meaning function* $T_{\mathcal{P}} : \mathcal{I}_{\mathcal{L}} \to \mathcal{I}_{\mathcal{L}}$ of P maps an interpretation I to the set of exactly those atoms A for which there is a clause $A \leftarrow \text{body} \in \mathcal{G}(P)$ such that the body is true under I. The operator T_P captures the semantics of P as the Herbrand models of the latter are exactly the pre-fixed points of the former, i.e., those interpretations I with $T_P(I) \subseteq I$. For logic programming purposes it is usually preferable to consider fixed points of T_P, instead of pre-fixed points, as the intended meaning of programs. These fixed points are called *supported models* [2]. The intended model M_2 for Example 3 is supported, while $\mathcal{B}_{\mathcal{L}}$ is a model but not supported.

A *level mapping* is a function assigning a natural number $|A| \geq 1$ to each ground atom A. For negative ground literals we define $|\neg A| := |A|$. A program P is called *acyclic* if there exists a level mapping $|\cdot|$ such that $|A| > |L_i|$ holds for all clauses $A \leftarrow L_1 \wedge \cdots \wedge L_n \in \mathcal{G}(P)$.

Example 5 Consider the program from Example 3 and let s^n denote the n-fold application of s. One possible level mapping for which we find that $\mathcal{P}_2$ is acyclic is given as $|\cdot| : \mathcal{B}_{\mathcal{L}} \to \mathbb{N}^+$ with $e(s^n(0)) \mapsto 2n+1$ and $o(s^n(0)) \mapsto 2n+2$. ⋆

In [14], the following metric was used to show the convergence of the iteration of $T_{\mathcal{P}}$ for acyclic logic programs. This can be done by showing that the space of all interpretations together with this metric is complete and that the $T_{\mathcal{P}}$-operator is contractive on it. Under these conditions we can apply Banach's contraction mapping theorem (see [36]), which allows to conclude the existence of a unique fixed point and to conclude that the iteration of $T_{\mathcal{P}}$ will converge to it.

Definition 6 *Let I and J be Herbrand interpretations. We define $d_{\mathcal{L}}(I, J) = 0$ for $I = J$ and $d_{\mathcal{L}}(I, J) = 2^{-n}$ if n is the smallest level on which I and J disagree.*

The set of all interpretations together with d comprises a compact and hence complete metric space (as shown in [14]). This insight will be applied later in Section 4.2.

Proposition 7 *$(\mathcal{I}_{\mathcal{L}}, d_{\mathcal{L}})$ is a compact metric space.*

Using level mappings, we can also define a notion of approximation. Here, an atom of smaller level is considered more important than one of higher level. By choosing a certain level mapping, we can specify the importance of the different atoms. We say that an interpretation I *approximates* an interpretation J *to degree* $n \in \mathbb{N}$ iff $d(I, J) \leq 2^{-n}$, i.e., if I and J agree on all atoms up to level n. This notion of approximation can be extended to $T_{\mathcal{P}}$-operators as follows:

Definition 8 *We say that an operator $\bar{T}_{\mathcal{P}}$ approximates the operator $T_{\mathcal{P}}$ to degree n, iff we find $d(\bar{T}_{\mathcal{P}}(I), T_{\mathcal{P}}(I)) \leq 2^{-n}$ for all interpretations $I \in \mathcal{I}_{\mathcal{L}}$.*

Definition 9 *Let $\mathcal{P}$ be an acyclic logic program wrt. some injective level mapping $|\cdot|$ and let $n \geq 0$. Then $\bar{\mathcal{P}} \subseteq \mathcal{G}(\mathcal{P})$ is defined as follows:*

$$\bar{\mathcal{P}} = \{H \leftarrow L_1 \wedge \ldots \wedge L_c \mid H \leftarrow L_1 \wedge \ldots \wedge L_n \in \mathcal{G}(\mathcal{P}) \text{ and } |H| \leq n\}$$

We will use $\bar{T}_{\mathcal{P}}$ to denote the operator associated with $\bar{\mathcal{P}}$.

The following lemma, shows the connection between an acyclic logic program $\mathcal{P}$ and its approximation $\bar{\mathcal{P}}$. In Section 4, we will embed this approximating operator into a connectionist system. A similar result is used, for example in [25].

Lemma 10 *Let $\mathcal{P}$ be an acyclic logic program wrt. some injective level mapping and let $\bar{\mathcal{P}}$ for $n \geq 0$ be defined as above. Then $\bar{\mathcal{P}}$ is finite and $\bar{T}_{\mathcal{P}}$ approximates $T_{\mathcal{P}}$ to degree n.*

Proof (Sketch) Because $\mathcal{P}$ is acyclic wrt. some injective level mapping, we find, that there are only finitely many atoms $H \in \mathcal{B}_{\mathcal{L}}$ with $|H| \leq n$, i.e., there are only finitely many different heads. Furthermore, we know that $|L_i| < |H|$, i.e., there are only finitely many body literals, which completes the proof of the first statement. For the second statement we first notice that obviously $T_{\mathcal{P}} = T_{\mathcal{G}(\mathcal{P})}$. Furthermore, $T_{\mathcal{G}(\mathcal{P})}(I)$ and $\bar{T}_{\mathcal{P}}(I)$ agree on all atoms with level $\leq n$, because all clauses with heads of level $\leq n$ are still contained in $\bar{\mathcal{P}}$. Therefore, we find that $d(\bar{T}_{\mathcal{P}}(I), T_{\mathcal{P}}(I)) \leq 2^{-n}$ for all interpretations $I \in \mathcal{I}_{\mathcal{L}}$. □

2.2. *Artificial Neural Networks*

Artificial neural networks consist of simple computational units (neurons), which receive real numbers as inputs via weighted connections and perform *simple* operations: the weighted inputs are added and simple functions (like threshold, sigmoidal) are applied to the sum. We will consider networks, where the units are organized in layers. Neurons which do not receive input from other neurons are called *input neurons*, and those without outgoing connections to other neurons are called *output neurons*. Such so-called *feed-forward networks* compute functions from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n and m are the number of input and output units, resp. The gray shaded area in Figure 1 on the left shows a simple feed-forward network with n input and n output units. In this paper we will construct recurrent networks by connecting the output units of such a feed-forward network to its input units.

3. The Core Method

In a nutshell, the idea behind the core method is to use feed-forward connectionist networks – called *core* – to compute or approximate the meaning function of logic programs. If the output layer of a core is connected to its input layer then these recurrent connections allow for an iteration of the meaning function. This iteration may converge to stable state, which corresponds to the desired (in some cases the least) model of the logic program (see Figure 1). Moreover, the core can be trained using standard methods from connectionist systems, in which case it can be conceived as acquiring a connectionist representation of logical knowledge. In other words, we are considering connectionist model generation using training and then iteration of recurrent networks with feed-forward core.

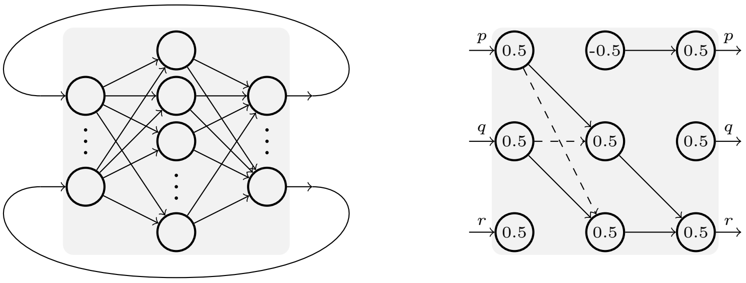


Figure 1. The blueprint of a recurrent network used by the core method and the core corresponding to $\mathcal{P}_1$. Solid connections have weight 1.0, dashed weight -1.0. The numbers denote the thresholds.

The ideas behind the core method for propositional logic programs were first presented in [21] (see also [19]). Consider the logic program from Example 2. A translation algorithm turns the program into a core of logical threshold units. Because the program contains the predicate letters p, q and r only, it suffices to consider interpretations of these three letters. Those can be represented by triples of logical threshold units. The input and the output layer of the core consist exactly of such triples. For each rule of the program a unit is added to the hidden layer such that the unit becomes active iff the preconditions of the rule are met by the current activation pattern of the input layer. This unit activates the output layer unit corresponding to the postcondition of the rule. The right part of Figure 1 shows the network obtained by the translation algorithm if applied to $\mathcal{P}_1$.

In [21] we proved – among other results – that for each propositional logic program $\mathcal{P}$ there exists a core computing its meaning function $T_{\mathcal{P}}$ and that for each acyclic propositional logic program $\mathcal{P}$ there exists a core with recurrent connections such that the computation with an arbitrary initial input converges to the unique fixed point of $T_{\mathcal{P}}$.

The use of logical threshold units in [21] made it easy to prove these results. However, it prevented the application of standard training methods like back-propagation to the kernel. This problem was solved in [13] by showing that the same results can be achieved if bipolar sigmoidal units are used instead. [13] also overcomes a restriction of the KBANN method originally presented in [35]: rules may now have arbitrarily many preconditions and programs may have arbitrarily many rules with the same postcondition.

The propositional core method has been extended in many directions. In [24] and [25] it was extended to finitely determined sets of truth values. Modal logic programs have been considered in [11]. Answer set programming and meta-level priorities are discussed in [10] and an application to intuitionistic logic programs in [12].

To summarize, the propositional core method allows for model generation with respect to a variety of logics in a connectionist setting. Given logic programs are translated into recurrent connectionist networks with feed-forward cores, such that the cores compute the meaning functions associated with the programs. The cores can be trained using standard learning methods leading to improved logic programs. These improved programs must be extracted from the trained cores in order to complete the neural-symbolic cycle. The extraction is outside the scope of this paper and interested readers are referred to [1,10].

While it was shown in [13], that the propositional core method leads to improved training behaviour, it is important to notice that propositional logic is in general insufficient for knowledge based intelligent systems. We thus carried it over to first-order logic programs, which is what we will report in the sequel.

4. Connectionist Representation of First-Order Programs

The extension of the core method to first-order logic poses a considerable problem because first-order interpretations usually do not map a finite but a countably infinite set of ground atoms to the set the truth values. Hence, they cannot be represented by a finite vector of units, each of which represents the value assigned to a particular ground atom. This is due to the fact that first-order logic programs allow to model infinite structures (such as natural numbers), while neural networks consist of only finitely many nodes. On the other hand, it is well known that multilayer feed-forward networks are universal approximators [20,16] for certain functions of the type $\mathbb{R}^n \to \mathbb{R}^m$. Hence, if we find a suitable way to represent interpretations of first-order logic programs by (finite vectors of) real numbers, then feed-forward networks can be used to approximate the meaning function of such programs. It is necessary, however, that such a representation is compatible with both, the logic programming and the neural network paradigm, as the resulting connectionist system shall combine desirable properties from both worlds, such as reasoning capabilities on the one hand, and robustness and trainability on the other hand. We thus require a way to bridge the gap between the discrete world of logic programs and the continuous world of neural networks. This will be realized by means of a *continuous* embedding in the sense of metric spaces. The rationale behind this is that the underlying metrics are meaningful for both logic programming and neural networks. On the technical side, following [22], we use level mappings to realize the representation.

Definition 11 *Let $I \in \mathcal{I}_\mathcal{L}$ be a Herbrand interpretation over $\mathcal{B}_\mathcal{L}$, $|\cdot|$ be an injective level mapping from $\mathcal{B}_\mathcal{L}$ to $\mathbb{N}^+$ and $b > 2$. Then we define the embedding function ι as follows:*

$$\iota : \mathcal{I}_\mathcal{L} \to \mathbb{R}, \; I \mapsto \sum_{A \in I} b^{-|A|}.$$

We will use $\mathfrak{C} := \{\iota(I) \mid I \in \mathcal{I}_\mathcal{L}\} \subset \mathbb{R}$ to denote the set of all embedded interpretations.

Definition 12 *Let $\mathcal{P}$ be a logic program and $T_\mathcal{P}$ its associated meaning operator. The embedding $f_\mathcal{P} : \mathfrak{C} \to \mathfrak{C}$ of $T_\mathcal{P}$ is defined as $f_\mathcal{P}(r) = \iota(T_\mathcal{P}(\iota^{-1}(r)))$.*

For $b > 2$, we find that the following mapping ι is injective from $\mathcal{I}_\mathcal{L}$ to $\mathbb{R}$ and, hence, bijective between $\mathcal{I}_\mathcal{L}$ and $\mathfrak{C}$. Please note that ι is not injective for $b = 2$, because $0.0\bar{1}_2 = 0.1_2$. In [22] we proved – among other results – that for each logic program $\mathcal{P}$ which is acyclic wrt. an injective level mapping the function $f_\mathcal{P}$ is continuous and contractive for $b > 3$. Moreover we find, that $\mathfrak{C}$ is a compact subset of the real numbers. This has various implications: (i) We can apply Funahashi's result, viz. that every continuous function on (a compact subset of) the reals can be uniformly approximated by feed-forward networks with sigmoidal units in the hidden layer [16]. This shows that the meaning function of acyclic logic program can be approximated by a core. (ii) Considering the metric from Definition 6, we can apply Banach's contraction mapping theorem [36] to conclude that the meaning function has a unique fixed point, which is obtained from an arbitrary initial interpretation by iterating the application of the meaning function. Using (i) and (ii) we were able to prove in [22] that the least model of logic programs which are acyclic wrt. a bijective level mapping can be approximated arbitrarily well by recurrent networks with feed-forward core. We will re-derive this as Theorem 21 below.

But what exactly is the approximation of an interpretation or a model in this context? Let $\mathcal{P}$ be a logic program and $|\cdot|$ be a level mapping. Using the metric introduced in Definition 6, we can use the notion of approximation as given in Definition 8. In other words, if a recurrent network approximates a model I of a logic program to a degree $n \in \mathbb{N}$ and outputs r then for all ground atoms A whose level is equal or less than n we find that $I(A) = \iota^{-1}(r)(A)$. All these relations will be discussed in detail below.

To summarize, from [22] we learn that there exists a recurrent network with a feed-forward core approximating the least fixed point of $T_{\mathcal{P}}$ for an acyclic logic program $\mathcal{P}$ arbitrarily well. But this result is purely theoretical and does not provide a way to actually construct the network. A constructive approach will be discussed in the sequel. In Section 4.1, we will show the underlying intuition by discussing an approach, which approximates the meaning functions of logic programs by means of piecewise constant functions $\bar{f}_{\mathcal{P}} : \mathbb{R} \to \mathbb{R}$, which will be implemented as a connectionist system. Unfortunately, we will find that the accuracy is very limited. This approach is then extended to a multi-dimensional setting in Section 4.2, allowing for arbitrary precision, even if implemented on a real computer. A novel training method, tailored for our specific setting is discussed in Section 5 and some preliminary experiments are presented in Section 6.

4.1. *The Underlying Intuition*

In this section, we will show how to construct a core network approximating the meaning operator of a given logic program. As mentioned above, this is not meant to be used in real implementations, but rather to convey the underlying intuitions. All details, including the proofs, will be given in the next section. As above, we will consider logic programs $\mathcal{P}$ which are acyclic wrt. a bijective level mapping. It suffices to require injectivity, but bijectivity allows for cleaner proofs. We will construct sigmoidal networks and RBF networks with a raised cosine activation function.

To illustrate the ideas, we will use the program $\mathcal{P}_2$ given in Example 3 as a running example. The construction consists of five steps: (i) Construct $f_{\mathcal{P}}$ as a real embedding of $T_{\mathcal{P}}$. (ii) Approximate $f_{\mathcal{P}}$ using a piecewise constant functions $\bar{f}_{\mathcal{P}}$. (iii) Implement $\bar{f}_{\mathcal{P}}$ using (a) step and (b) triangular functions. (iv) Implement $\bar{f}_{\mathcal{P}}$ using (a) sigmoidal and (b) raised cosine functions. (v) Construct the core network approximating $f_{\mathcal{P}}$.

In the sequel we will describe the ideas underlying the construction. The more general m-dimensional approach will be described in the following sections. One should observe that $f_{\mathcal{P}}$ is a function on $\mathfrak{C}$ and not on $\mathbb{R}$. Although the functions constructed below will be defined on intervals of $\mathbb{R}$, we are thus concerned with accuracy on $\mathfrak{C}$ only.

Construct $f_{\mathcal{P}}$ as a real embedding of $T_{\mathcal{P}}$: We use $f_{\mathcal{P}}(r) = \iota(T_{\mathcal{P}}(\iota^{-1}(r)))$ as a real-valued version for $T_{\mathcal{P}}$. But first, we will have a closer look at its domain $\mathfrak{C}$. For readers familiar with fractal geometry, we note that $\mathfrak{C}$ is a variant of the classical Cantor set [8]. The interval $[0, \frac{1}{b-1}]$ is split into b equal parts, where b is the natural number used in the definition of the embedding function ι. All, except the left- and rightmost subintervals are removed. The remaining two parts are split again and the subintervals except the first and the last are removed, etc. We use $\mathfrak{C}_n$ to denote the result after n splits and removals. This process is repeated ad infinitum and we find $\mathfrak{C}$ to be its limit, i.e., $\mathfrak{C} = \bigcap_{i \in \mathbb{N}} \mathfrak{C}_i$. The first four iterations are depicted in Figure 2.

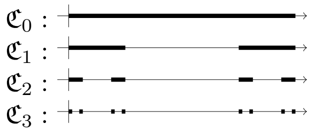


Figure 2. The first four iterations ($\mathfrak{C}_0$, $\mathfrak{C}_1$, $\mathfrak{C}_2$ and $\mathfrak{C}_3$) of the construction of $\mathfrak{C}$ for $b = 4$.

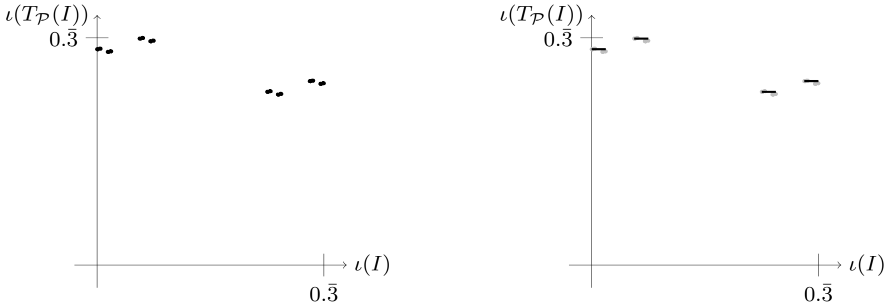


Figure 3. The plot of $f_{\mathcal{P}_2}$ is shown on the left. On the right a piecewise constant approximation $\bar{f}_{\mathcal{P}_2}$ (for level $n = 3$) of $f_{\mathcal{P}_2}$ (depicted in light gray) is shown.

Example 13 Considering program $\mathcal{P}_2$, and the level mapping from Example 5, we obtain $f_{\mathcal{P}_2}$ as depicted in Figure 3 on the left. ⋆

Approximate $f_{\mathcal{P}}$ using a piecewise constant function $\bar{f}_{\mathcal{P}}$: Under the assumption that $\mathcal{P}$ is acyclic, we find that all variables occurring in the body of a clause are also contained in its head. Hence, for each level n and two interpretations I and J, we find that whenever $d(I, J) \leq 2^{-n}$ holds, $d(T_{\mathcal{P}}(I), T_{\mathcal{P}}(J)) \leq 2^{-n}$ follows. Therefore, we can approximate $T_{\mathcal{P}}$ to degree n by some function $\bar{T}_{\mathcal{P}}$ which considers ground atoms with a level less or equal to n only. Due to the acyclicity of $\mathcal{P}$, we can construct a finite ground program $\bar{\mathcal{P}} \subseteq \mathcal{G}(\mathcal{P})$ containing those clauses of $\mathcal{G}(\mathcal{P})$ with literals of level less or equal n only and find $T_{\bar{\mathcal{P}}} = \bar{T}_{\mathcal{P}}$ (see Lemma 10). We will use $\bar{f}_{\mathcal{P}}$ to denote the embedding of $\bar{T}_{\mathcal{P}}$ and we find that it is a piecewise constant function, being constant on each connected interval of $\mathfrak{C}_{n-1}$. Furthermore, we find that $|f_{\mathcal{P}}(x) - \bar{f}_{\mathcal{P}}(x)| \leq 2^{-n}$ for all $x \in \mathfrak{C}$, i.e., $\bar{f}_{\mathcal{P}}$ is a constant piecewise approximation of $f_{\mathcal{P}}$. As mentioned above, we will discuss all this formally in Section 4.2.

Example 14 For our running example $\mathcal{P}_2$ and $n = 3$, we obtain $\bar{\mathcal{P}}_2 = \{e(0).\ e(s(0)) \leftarrow o(0).\ o(0) \leftarrow \neg e(0).\}$ and $\bar{f}_{\mathcal{P}_2}$ as depicted in Figure 3 on the right. ⋆

Implement $\bar{f}_{\mathcal{P}}$ using (a) step and (b) triangular functions: As a next step, we will show how to implement $\bar{f}_{\mathcal{P}}$ using step and triangular functions. Those functions are the linear counterparts of the functions actually used in the networks constructed below. If $\bar{f}_{\mathcal{P}}$ consists of k intervals, then we can implement it using $k - 1$ step functions which are placed such that the steps are between two neighboring intervals. This is depicted in Figure 4 on the left.

Each constant piece of length $\lambda := \frac{1}{b-1} \times \frac{1}{b^n}$ could also be implemented using two triangular functions with width λ which are centered at the endpoints. Those two triangles

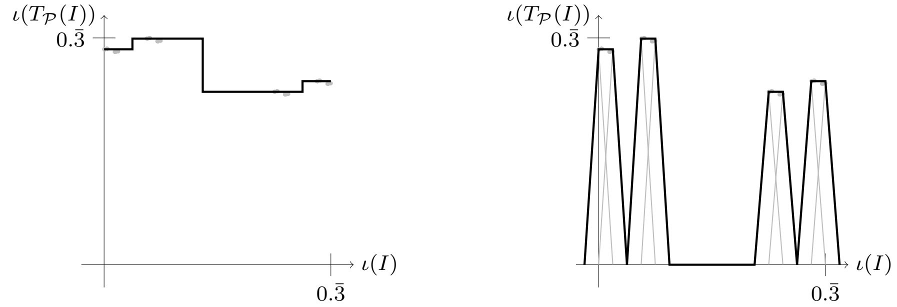


Figure 4. Two linear approximation of $\bar{f}_{\mathcal{P}_2}$. On the left, three step functions were used; On the right, eight triangular functions (shown in gray) add up to the approximation, which is shown using thick lines.

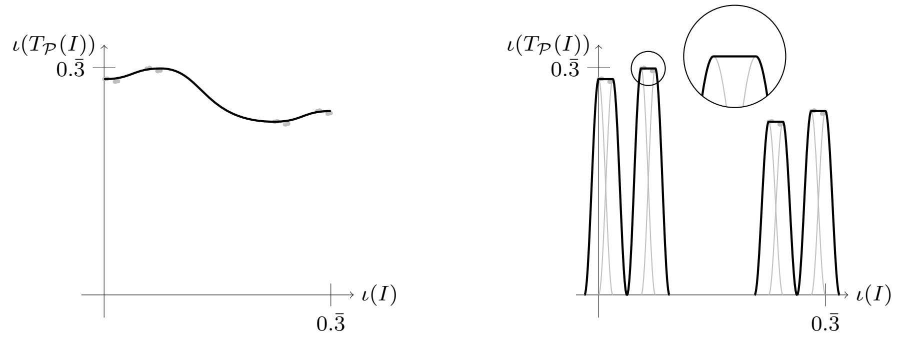


Figure 5. Two non-linear approximation of $\bar{f}_{\mathcal{P}_2}$. On the left, sigmoidal functions were used and on the right, raised cosines.

add up to the constant piece. For base b, we find that the gaps between two intervals have a length of at least $(b-2)\lambda$. Therefore, the triangular functions of two different intervals will never interfere. This is depicted in Figure 4 on the right.

Implement $\bar{f}_\mathcal{P}$ using (a) sigmoidal and (b) raised cosine functions: To obtain a sigmoidal approximation, we replace each step function with a sigmoidal function. Unfortunately, those add some further approximation error, which can be dealt with by increasing the accuracy in the constructions above. By dividing the desired accuracy by two, we can use one half as accuracy for the constructions so far and the other half as a margin to approximate the constant pieces by sigmoidal functions. This is possible because we are concerned with the approximation on $\mathfrak{C}$ only. The triangular functions can simply be replaced by raised cosine activation functions, as those add up exactly as the triangles do and do not interfere with other intervals either.

Construct the core network approximating $f_\mathcal{P}$: A sigmoidal network approximating a given $T_\mathcal{P}$-operator consists of: An input layer containing one input unit whose activation will represent an interpretation I. A hidden layer containing a unit with sigmoidal activation function for each sigmoidal function constructed above. An output layer containing one unit whose activation will represent the approximation of $T_\mathcal{P}(I)$. The weights from input to hidden layer together with the bias of the hidden units define the positions of the sigmoidals. The weights from hidden to output layer represent the heights of the single functions. An RBF network can be constructed analogously, but will contain more hidden layer units, namely one for each raised cosine functions. A constructive proof for the existence of approximating networks follows directly from the fact that the network constructed above approximates a given $T_{\mathcal{P}}$-operator up to any given accuracy.

4.2. *A Distributed Embedding*

In the previous section, we showed how to construct a core network for a given program and some desired level of accuracy. Due to the one-dimensional embedding, the precision of a real implementation is very limited. This limitation can be overcome by distributing an interpretation over more than one real number. In our running example $\mathcal{P}_2$, we could embed all *even*-atoms into one real number and all *odd*-atoms into another one, thereby obtaining a two-dimensional vector for each interpretation, hence doubling the accuracy. By embedding interpretations into higher-dimensional vectors, we can approximate meaning functions of logic programs arbitrarily well.

We now present the details of this approach. We base our treatment on RBF networks, although in principle other kinds of activation functions could also be used. Our choice was in part inspired by vector-based networks as described in [15]. Analogously to the previous section, we will proceed as follows: (i) Construct $f_{\mathcal{P}}$ as a real embedding of $T_{\mathcal{P}}$. (ii) Approximate $f_{\mathcal{P}}$ using a piecewise constant functions $\bar{f}_{\mathcal{P}}$. (iii) Construct the core network approximating $f_{\mathcal{P}}$. After discussing a new embedding of interpretations into vectors of real numbers, we will show how to approximate the embedded $T_{\mathcal{P}}$-operator using a piecewise constant function. This piecewise function will then be implemented using a connectionist system. The system presented here is a *fine blend* of ideas from *vector-based networks* and the approach presented above.

Construct $f_{\mathcal{P}}$ as a real embedding of $T_{\mathcal{P}}$: We will first extend level mappings to a multi-dimensional setting, and then use them to represent interpretations as real vectors. This leads to a new embedding of $T_{\mathcal{P}}$.

Definition 15 *An m-*dimensional level mapping $\|\cdot\| : \mathcal{B}_{\mathcal{L}} \to (\mathbb{N}^+, \{1, \ldots, m\})$ *maps atoms to two natural numbers. For $A \in \mathcal{B}_{\mathcal{L}}$ and $\|A\| = (l, d)$, we call l and d the* level *and* dimension *of A respectively. Again, we define $\|\neg A\| := \|A\|$.*

Example 16 A possible 2-dimensional level mapping for $\mathcal{P}_2$ is given as:

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

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

Definition 17 *Let $b \geq 3$ and let $A \in \mathcal{B}_{\mathcal{L}}$ be an atom with $\|A\| = (l, d)$. The m-*dimensional embedding $\boldsymbol{\iota}{:}\mathcal{B}_{\mathcal{L}} \to \mathbb{R}^m$ *and its extensions $\boldsymbol{\iota}{:}\mathcal{I}_{\mathcal{L}} \to \mathbb{R}^m$ are defined as:*

$$\boldsymbol{\iota}(A) := (\iota_1(A), \ldots, \iota_m(A)) \quad \textit{with} \quad \iota_j(A) := \begin{cases} b^{-l} & \textit{if } j = d \\ 0 & \textit{else} \end{cases} \quad \textit{and} \quad \boldsymbol{\iota}(I) := \sum_{A \in I} \boldsymbol{\iota}(A).$$

We will use $\mathfrak{C}^m := \{\boldsymbol{\iota}(I) \mid I \in \mathcal{I}_{\mathcal{L}}\} \subset \mathbb{R}^m$ to denote the set of all embedded interpretations.

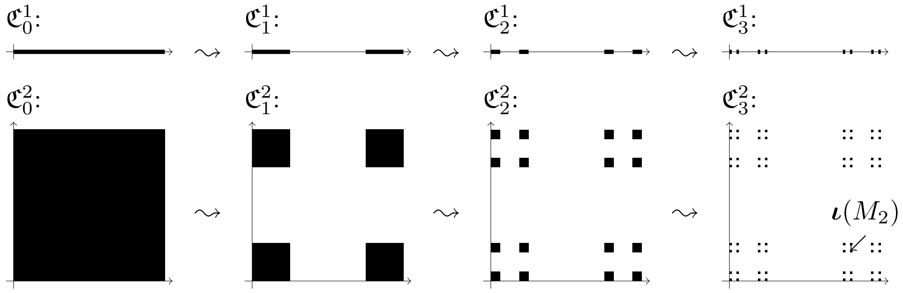


Figure 6. The first four iterations of the construction of $\mathfrak{C}^1$ are depicted on the top. The construction of $\mathfrak{C}^2$ is depicted below containing $\boldsymbol{\iota}(M_2)$ from Example 18. Both are constructed using $b = 4$.

As mentioned above, $\mathfrak{C}^1$ is the classical Cantor set and $\mathfrak{C}^2$ the 2-dimensional variant of it [8]. Without going into detail, Figure 6 shows the first 4 steps in the construction of $\mathfrak{C}^2$. The big square is first replaced by 2^m shrunken copies of itself, the result is again replaced by 2^m smaller copies and so on. The limit of this iterative replacement is $\mathfrak{C}^2$. Again, we will use $\mathfrak{C}^m_i$ to denote the result of the i-th replacement, i.e., Figure 6 depicts $\mathfrak{C}^2_0, \mathfrak{C}^2_1, \mathfrak{C}^2_2$ and $\mathfrak{C}^2_3$. For readers with a background in fractal geometry we note that these are the first four applications of an appropriately set up iterated function system [8].

Example 18 Using the 1-dimensional level mapping from Example 5, we obtain $\mathfrak{C}^1$ as depicted in Figure 6 at the top. Using the 2-dimensional from above, we obtain $\mathfrak{C}^2$ and $\boldsymbol{\iota}(M_2) = (0.10\overline{10}_b, 0.01\overline{01}_b) \approx (0.2666667, 0.0666667)$ for the embedding of the intended model M_2. $\star$

Next, we will discuss some important properties of the embedding $\boldsymbol{\iota}$ and its codomain $\mathfrak{C}^m$. These include the bijectivity and continuity of $\boldsymbol{\iota}$, as well as the compactness of $\mathfrak{C}^m$. The bijectivity allows to relate the $T_{\mathcal{P}}$-operator and its embedded version in a precise way. Furthermore, we will establish a precise relation between the space of interpretations and their embeddings. We will consider $\boldsymbol{\iota}$ as a mapping from $\mathcal{B}_{\mathcal{L}}$ to $\mathfrak{C}^m$.

Lemma 19 *The following three properties hold:*
(i) $\boldsymbol{\iota}$ is a bijection between $\mathcal{B}_{\mathcal{L}}$ and $\mathfrak{C}^m$.
(ii) $\boldsymbol{\iota} : \mathcal{B}_{\mathcal{L}} \to \mathbb{R}^m$ is a continuous mapping from $(\mathcal{I}_{\mathcal{L}}, d_{\mathcal{L}})$ to $(\mathbb{R}^m, d_{\mathbb{R}})$.
(iii) $\boldsymbol{\iota}^{-1} : \mathfrak{C}^m \to \mathcal{B}_{\mathcal{L}}$ is also a continuous mapping.

Proof First we show that $\boldsymbol{\iota}$ is injective (one-to-one), 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 $A \in \mathcal{B}_{\mathcal{L}}$ 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 $< d$ on which both disagree. Let $K := \{B \in I \mid \|B\| = (l_B, d_B) \text{ and } d_B < d\}$, 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 $J' := J \setminus (\{A\} \cup K)$. Looking at dimension d only, we find $\boldsymbol{\iota}_d(I) = \boldsymbol{\iota}_d(K) + b^{-d} + \boldsymbol{\iota}_d(I')$ and $\boldsymbol{\iota}_d(J) = \boldsymbol{\iota}_d(K) + \boldsymbol{\iota}_d(J')$. I.e., to derive a contradiction it suffices to show that $\boldsymbol{\iota}_d(J') < b^{-d}$, which follows from the fact that $\boldsymbol{\iota}_d(J')$ can be at most $\frac{b^{-d}}{b-1}$ which is smaller than b^{-d}. Obviously, $\boldsymbol{\iota}$ is also surjective (onto), hence it is bijective.

To show the continuity of $\boldsymbol{\iota}$, we need to show that there is a $\delta > 0$ for all $\varepsilon > 0$, such that $d_{\mathcal{L}}(I, J) < \delta$ implies $d_{\mathbb{R}}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) < \varepsilon$. As in the proof of injectivity just given, we assume that I and J differ on some atom A (with $A \in I$ and $A \notin J$) of level n and agree on all atoms with smaller level. For I' and J' as defined above, we get $d_{\mathcal{L}}(I, J) = b^{-n}$ and $d_{\mathbb{R}}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) = b^{-n} + \boldsymbol{\iota}_d(I') - \boldsymbol{\iota}_d(J') < b^{-n} + \frac{b^{-d}}{b-1}$. Obviously, there exists an n for each $\varepsilon > 0$ such that $b^{-n} + \frac{b^{-d}}{b-1} < \varepsilon$ and for $\delta = b^{-n}$, we find $d_{\mathbb{R}}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) < b^{-n} + \frac{b^{-d}}{b-1} < \varepsilon$ for all I and J with $d_{\mathcal{L}}(I, J) < \delta$.

Analogously to the proof of continuity of $\boldsymbol{\iota}$ just given, we need to show that there is a $\delta > 0$ for all $\varepsilon > 0$, such that $d_{\mathbb{R}}(\boldsymbol{\iota}(I), \boldsymbol{\iota}(J)) < \delta$ implies $d_{\mathcal{L}}(I, J) < \varepsilon$. As in the proof above, this can be done by computing a suitable n. □

In other words, Lemma 19 states that ι is a *homeomorphism*, i.e., a structure-preserving mapping between two metric spaces. Homeomorphisms are fundamental to theory of metric spaces in that central notions such as compactness and continuity are preserved under them. With general knowledge from metric space theory (see [36]), we obtain:

Corollary 20 *$\mathfrak{C}^m$ is a compact subset of $\mathbb{R}^m$. Furthermore, $T_{\mathcal{P}}$ is continuous if and only if $\boldsymbol{\iota}(T_{\mathcal{P}})$ is continuous.*

Proof The statements follow from the fact that homeomorphisms preserve compactness and continuity in the way needed for the result (see [36]). □

We see from these results that ι indeed serves as a bridge between the space of interpretations of a logic program, and the reals. Obviously, the natural metric on the reals – which underlies the standard notion of continuity on the reals – is a meaningful notion for the continuous neural network paradigm. The question remains, though, whether the considered metric on the space of all interpretations is meaningful for logic programs. And indeed, this metric is most suitable for capturing semantic notions for logic programs, which is well-known from research in this area (see [14,31]). Our embedding ι is thus a priori a promising candidate for a useful connectionist representation of logic programs, and we will indeed verify this in the sequel.

Knowing that $\mathfrak{C}^m$ is a compact subset of the reals, we now immediately obtain the main theorem of [22], stating that there exists an approximating network for a given acyclic program $\mathcal{P}$. It follows from the fact that $T_{\mathcal{P}}$ is continuous for acyclic programs. Using Corollary 20 we can also conclude that $f_{\mathcal{P}}$ is a continuous function on a compact set, which allows to apply Funahashi's theorem [16].

Theorem 21 *Let $\mathcal{P}$ be acyclic. Then there exists a 3-layer connections system which approximates $T_{\mathcal{P}}$ arbitrarily well.*

Using the m-dimensional embedding, the $T_{\mathcal{P}}$-operator can be embedded into real vectors to obtain a real-valued function $f_{\mathcal{P}}$. The following theorem follows immediately from the fact that $\boldsymbol{\iota}$ is a bijection.

Definition 22 *Let $\boldsymbol{\iota}$ be an m-dimensional embedding. The* m-dimensional embedding *$f_{\mathcal{P}}$ of a given $T_{\mathcal{P}}$-operator is defined as $f_{\mathcal{P}} : \mathfrak{C}^m \to \mathfrak{C}^m : \boldsymbol{x} \mapsto \boldsymbol{\iota}\left(T_{\mathcal{P}}\left(\boldsymbol{\iota}^{-1}(\boldsymbol{x})\right)\right)$.*

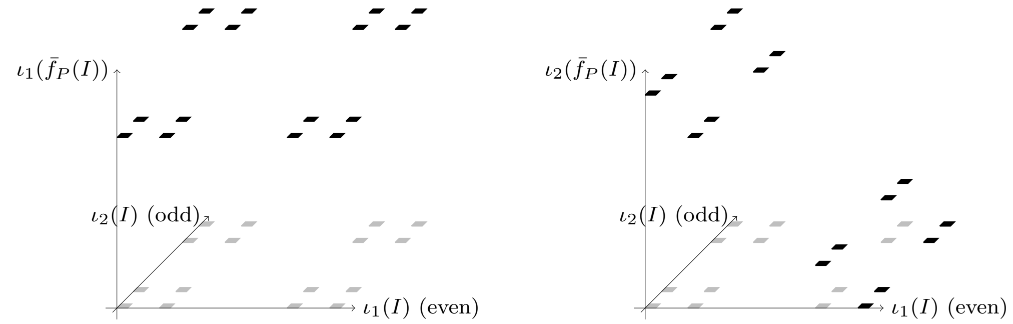


Figure 7. $\bar{f}_{\mathcal{P}}$ for $\mathcal{P}_2$, the 2-dimensional level mapping from Example 16 and $n = 3$. The outputs for dimension 1 and 2 are shown on the left and on the right, respectively.

Theorem 23 *Let $T_{\mathcal{P}}$ be a consequence operator defined over $\mathcal{B}_{\mathcal{L}}$, $\mathcal{I}_{\mathcal{L}}$ be the corresponding space of interpretations and let $\boldsymbol{\iota} : \mathcal{B}_{\mathcal{L}} \to \mathbb{R}^m$ be the bijective m-dimensional embedding as introduced above. Then $T_{\mathcal{P}}(I) = \boldsymbol{\iota}^{-1}(f_{\mathcal{P}}(\boldsymbol{\iota}(I)))$.*

Approximate $f_{\mathcal{P}}$ using a piecewise constant function $\bar{f}_{\mathcal{P}}$: Under the assumption that $\mathcal{P}$ is acyclic, we can approximate $T_{\mathcal{P}}$ up to some level n by $\bar{T}_{\mathcal{P}}$. After embedding $\bar{T}_{\mathcal{P}}$ into $\mathbb{R}^m$, we find that it is constant on certain regions, namely on all those embedded interpretations which are contained in the same connected interval in $\mathfrak{C}_n^m$. Those intervals will be referred to as *hyper-squares* in the sequel. We will use H_l to denote a hyper-square of level l, i.e., one of the squares occurring in $\mathfrak{C}_l^m$. An approximation of $T_{\mathcal{P}}$ up to some level n will yield a function $\bar{f}_{\mathcal{P}}$ which is constant on all hyper-squares of level n.

Example 24 Considering program $\mathcal{P}_2$, and the level mapping from Example 16, we obtain $\bar{f}_{\mathcal{P}}$ for $n = 3$ as depicted in Figure 7. ⋆

Using the results discussed in Section 2.1, we can prove the following lemma, which provides the basis for the approximation of acyclic logic programs. It shows, that the $\bar{T}_{\mathcal{P}}$-operator for some n and, hence, its embedded version $\bar{f}_{\mathcal{P}}$ are constant on certain regions of the input space.

Lemma 25 *Let $\mathcal{P}$ be acyclic wrt. some injective level mapping $\| \cdot \|$, let $n \geq 0$ and let $\bar{T}_{\mathcal{P}}$ be as in Definition 9. Then we find $\bar{T}_{\mathcal{P}}(I) = \bar{T}_{\mathcal{P}}(J)$ and hence $\bar{f}_{\mathcal{P}}(\boldsymbol{\iota}(I)) = \bar{f}_{\mathcal{P}}(\boldsymbol{\iota}(J))$ for all $I, J \in \mathcal{I}_{\mathcal{L}}$ with $d_{\mathcal{L}}(I, J) \leq 2^{-n}$.*

Proof From $d_{\mathcal{L}}(I, J) \leq 2^{-n}$, we can conclude, that I and J agree on all atoms with level $\leq n$. Furthermore, we know that $\bar{T}_{\mathcal{P}}$ does only consider those atoms. Therefore, we can conclude that $\bar{T}_{\mathcal{P}}(I) = \bar{T}_{\mathcal{P}}(J)$. From the bijectivity of $\boldsymbol{\iota}$ we obtain immediately $\bar{f}_{\mathcal{P}}(\boldsymbol{\iota}(I)) = \bar{f}_{\mathcal{P}}(\boldsymbol{\iota}(J))$ for all $I, J \in \mathcal{I}_{\mathcal{L}}$ with $d_{\mathcal{L}}(I, J) \leq 2^{-n}$. □

Taking a closer look at the set of embedded interpretations whose distance is below 2^{-n}, we find that they are all contained in one of the hyper-squares of level n. First we will define the smallest and the largest interpretation, $s_n(I)$ and $l_n(I)$ respectively, with $d_{\mathcal{L}}(I, s_n(I)) \leq 2^{-n}$, $d_{\mathcal{L}}(I, l_n(I)) \leq 2^{-n}$ and $d_{\mathcal{L}}(s_n(I), l_n(I)) \leq 2^{-n}$. Afterwards, the corresponding hypersquares are defined.

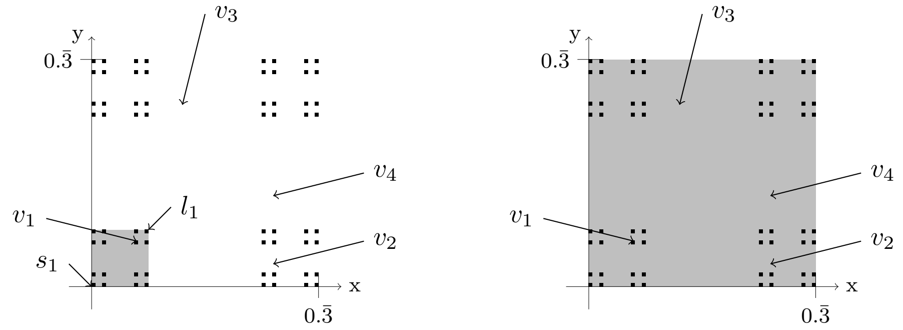


Figure 8. The largest exclusive hyper-square of v_1 with respect to the set $\{v_2, v_3, v_4\}$ is shown on the left, together with the smallest s_1 and largest l_1 embedded interpretation with respect to $\boldsymbol{\iota}^{-1}(v_1)$ and level 1. The smallest inclusive hyper-square of the set $\{v_1, v_2, v_3, v_4\}$ is shown on the right. Both are depicted in light gray together with $\mathfrak{C}^2$ in black.

Definition 26 *Let $I \in \mathcal{I}_{\mathcal{L}}$ and $n \geq 0$. We define smallest $s_n(I)$ and largest interpretation $l_n(I)$, and the corresponding hyper-square $H_n(I)$ of level n as follows:*

$$\begin{aligned} s_n(I) &:= \{A \in I \mid \|A\| = (l,d) \textit{ with } l \leq n\} \\ l_n(I) &:= \{A \in I \mid \|A\| = (l,d) \textit{ with } l \leq n\} \cup \{A \in \mathcal{B}_{\mathcal{L}} \mid \|A\| = (l,d) \textit{ with } l > n\} \\ H_n(I) &:= \prod_{1 \leq i \leq m} [\iota_i(s_n(I)), \iota_i(l_n(I))]. \end{aligned}$$

Example 27 Figure 8 on the left, shows the smallest s_1 and the largest l_1 embedded interpretation with respect to $\boldsymbol{\iota}^{-1}(v_1)$ and level 1. The corresponding hyper-square of level 1 is depicted in light gray. ⋆

Without proving it formally, we find $H_n(I) = H_n(J)$ for all interpretations I and J with $d_{\mathcal{L}}(I, J) \leq 2^{-n}$. Furthermore, we find that every such hyper-square is one of the connected intervals occurring in the n-th step of the construction of $\mathfrak{C}$. To be more precise, we find $\mathfrak{C}_n^m = \bigcup_{I \in \mathcal{I}_{\mathcal{L}}} H_n(I)$.

Construct the core network approximating $f_{\mathcal{P}}$: We will use a 3-layered network with a winner-take-all hidden layer. For each hyper-square H of level n, we add a unit to the hidden layer, such that the input weights encode the position of the center of H. The unit shall output 1 if it is selected as winner, and 0 otherwise. The weight associated with the output connections of this unit is the value of $\bar{f}_{\mathcal{P}}$ on that hyper-square. Thus, we obtain a connectionist network approximating the semantic operator $T_{\mathcal{P}}$ up to the given accuracy ε. To simplify the notations later, we will introduce the *largest exclusive hyper-square* and the *smallest inclusive hyper-square* as follows.

Definition 28 *The* largest exclusive hyper-square *of a vector $\boldsymbol{u} \in \mathfrak{C}_0^m$ and a set of vectors $V = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\} \subseteq \mathfrak{C}_0^m$, denoted by $H_{ex}(\boldsymbol{u}, V)$, either does not exist or is the hyper-square H of least level for which $\boldsymbol{u} \in H$ and $V \cap H = \emptyset$. The* smallest inclusive hyper-square *of a non-empty set of vectors $U = \{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k\} \subseteq \mathfrak{C}_0^m$, denoted by $H_{in}(U)$, is the hyper-square H of greatest level for which $U \subseteq H$.*

Example 29 Let $v_1 = (0.07, 0.07)$, $v_2 = (0.27, 0.03)$, $v_3 = (0.13, 0.27)$ and $v_4 = (0.27, 0.13)$ as depicted in Figure 8. The largest exclusive hyper-square of v_1 with respect to $\{v_2, v_3, v_4\}$ is shown in light gray on the left. That of v_3 with respect to $\{v_1, v_2, v_4\}$ does not exists, because there is no hyper-square which contains only v_3. The smallest inclusive hyper-square of all four vectors is shown on the right, and is in this case $\mathfrak{C}_0$. $\star$

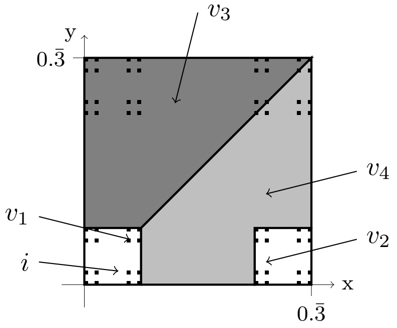
To determine the winner for a given input, we designed an activation function such that its outcome is greatest for the "responsible" unit. Those are defined as follows: Given some hyper-square H, units which are positioned in H but not in any of its sub-hyper-squares are called *default units* of H, and they are responsible for inputs from H except for inputs from sub-hyper-squares containing other units. If H does not have any default units, the units positioned in its sub-hyper-squares are responsible for all inputs from H as well. After all units' activations have been computed, the unit with the greatest value is selected as the winner. The details of this activation function $d_{\mathfrak{C}}$ are given in Algorithm 1. Please note that the algorithm outputs a 3-tuple, which is compared component wise, i.e., the first component is most important. If for two distances this first component is equal, the second component is used and, finally, the third, if the first two are equal. This activation function can easily be turned into a function computing a single real value.

Algorithm 1: The activation function for the Fine Blend

```
Input: Inputs $\boldsymbol{x}, \boldsymbol{y} \in \mathfrak{C}_0^m$
Output: Distance $d_{\mathfrak{C}}(\boldsymbol{x}, \boldsymbol{y}) \in (\mathbb{N}, \{1,2,3\}, \mathbb{R})$
if $\boldsymbol{x} = \boldsymbol{y}$ then return $(\infty, 0, 0)$
$l :=$ level of $H_{in}(\{\boldsymbol{x}, \boldsymbol{y}\})$
$k := \begin{cases} 3 & \text{if } H_{in}(\{\boldsymbol{x}\}) \text{ and } H_{in}(\{\boldsymbol{y}\}) \text{ are of level greater than } l \\ 2 & \text{if } H_{in}(\{\boldsymbol{x}\}) \text{ or } H_{in}(\{\boldsymbol{y}\}) \text{ is of level greater than } l \\ 1 & \text{otherwise} \end{cases}$
$m := \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|}$, i.e., $m$ is the inverse of the Euclidean distance
return $(l, k, m)$
```

Example 30 Let v_1, v_2, v_3 and v_4 from Example 29 be the incoming connections for some units. The different input regions for which each of the units are responsible together to their distances to the vector $i = (0.05, 0.02)$ are depicted in Figure 9. Here, v_1 and i are in the same hyper-square of level 1, one of them is located in one of the sub-hyper-squares and their Euclidean distance is $\frac{1}{20.18}$. We find $d_{\mathfrak{C}}(v_1, i) > d_{\mathfrak{C}}(v_4, i) > d_{\mathfrak{C}}(v_3, i) > d_{\mathfrak{C}}(v_2, i)$. Even though, v_2 is euclidically closer to i than v_3 and v_4 it is further away according to our distance function, because it is considered to be the default unit for the south-east hyper-square, whereas v_3 and v_4 are responsible for parts of the big square. $\star$

Now, we can state an algorithm that constructs a connectionist system for a given acyclic program and some level of accuracy n. This is done as Algorithm 2. First, m units are added to the input and output layer, one for each dimension of the embedding, and the hidden layer and the connections are initialized to be empty. Afterwards, the set of important atoms is constructed containing all those atoms with a level smaller than n. For each possible subset of those units, treated as an interpretation, we add a hidden layer



$$
\begin{aligned}
d_{\mathfrak{C}}(v_1, i) &= (1, 2, 20.18)\\
d_{\mathfrak{C}}(v_2, i) &= (0, 1, 4.61)\\
d_{\mathfrak{C}}(v_3, i) &= (0, 2, 3.84)\\
d_{\mathfrak{C}}(v_4, i) &= (0, 2, 4.09)
\end{aligned}
$$

Figure 9. The areas of responsibility for v_1, v_2, v_3 and v_4 and their distances to i.

unit, such that its incoming connections encode the center of the corresponding hyper-square and its outgoing connections encode the desired output value. As we will use a winner-take-all behaviour, only one of the input units will output 1, which is propagated through those connections, leading to the desired values in the output layer.

Algorithm 2: Construction of a Fine Blend-network for a given program.

```
Input: A program $\mathcal{P}$ which is acyclic wrt. an injective $m$-dimensional level mapping
       $\|\cdot\|$ and a level $n$
Output: A network, which computes the embedding of $T_{\mathcal{P}}$ up to level $n$
Let $U_i := \{I_1, \dots, I_m\}$, be a set of $m$ input units
Let $U_o := \{O_1, \dots, O_m\}$, be a set of $m$ output units with the identity as output
  function and the weighted sum as input function
Let $U_h := \{\}$, be an initially empty hidden layer, with winner-take-all behaviour,
  such that the winner output 1 and all other units output 0, using the activation
  function as detailed in Algorithm 1
Let $p := \{A \in \mathcal{B}_{\mathcal{L}} \mid \|A\| = (l, d) \text{ and } l \leq n\}$ be the set of important atoms
Let $C_{ih}$ and $C_{ho}$ be initially empty sets of connections
forall $I \subseteq p$ do
    Let $H_n(I)$ be the corresponding hyper-square for $I$ with center $c$
    Let $r := \boldsymbol{\iota}(T_{\mathcal{P}}(I))$ be the embedded desired result
    Add a hidden unit $h$ to $U_h$
    forall $1 \leq i \leq m$ do
        Connect $I_i$ to $h$ with weight $c_i$, i.e., add $I_i \xrightarrow{c_i} h$ to $C_{ih}$
        Connect $h$ to $O_i$ with weight $r_i$, i.e., add $h \xrightarrow{r_i} O_i$ to $C_{ih}$
return $\mathcal{N}_{\mathcal{P}} := (U_i, C_{ih}, U_h, C_{ho}, U_o)$
```

Example 31 Considering program $\mathcal{P}_2$, and the level mapping from Example 16, we obtain for $n = 2$ the connectionist systems depicted in Figure 10. Using e.g., $(0, 0)$ as input, we will find the following activation values for the hidden layer units: $h_1 \mapsto (1, 2, 70.7)$, $h_{2/3} \mapsto (0, 3, 3.84)$ and $h_4 \mapsto (0, 3, 2.72)$. Therefore, unit h_1 would be the winner and hence the output will be $(0.25, 0.25)$. This corresponds to the application of $\bar{T}_{\mathcal{P}}$ to the empty interpretation, which yields $\{e(0), o(0)\}$. ⋆

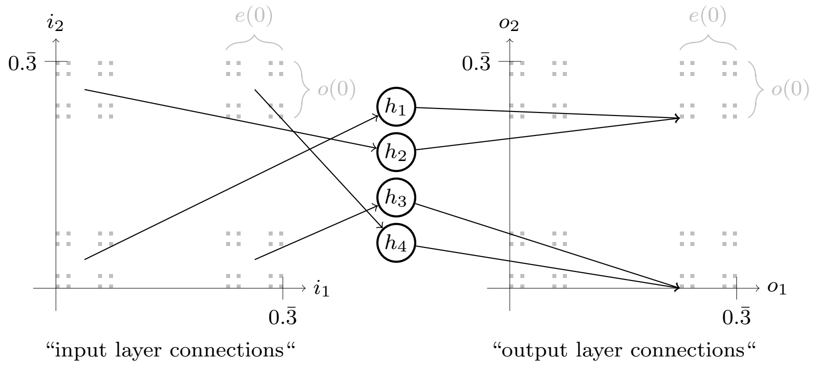


Figure 10. The connectionist system corresponding to $\mathcal{P}_2$ for $n = 2$. Instead of showing the input and output units and the values of the weights, the positions are indicated. There are actually two input and two output units. If both input units would be activated with 0.0, the hidden unit h_1 would become active (as the closest to this input), and hence, the output would be $(0.25, 0.25)$. For all inputs from the right side of the input space ($e(0)$ is true), i.e., for those that activate h_3 and h_4, we would obtain an output of $(0.25, 0)$ which corresponds to the conclusion that only $e(0)$ is true.

5. Learning

In this section, we will describe the adaptation of the system 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, or to train a network from scratch, i.e., using an initial network with a single hidden layer unit. The training samples in our case come from the original (non approximated) program, but might also be observed in the real world or given by experts. 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 [15]. In particular, we used the notion of utility as used for the refinement of vector-based neural networks.

Adapting the weights Let $\boldsymbol{x}$ be the input, $\boldsymbol{y}$ be the desired output and u be the winner-unit from the hidden layer. Let $\boldsymbol{w}_{in}$ denote the weights of the incoming connections of u and $\boldsymbol{w}_{out}$ be the weights of the outgoing connections. To adapt the system, we move u towards the center $\boldsymbol{c}$ of the smallest inclusive hyper-square $H_{in}(\{\boldsymbol{x}, u\})$:

$$\boldsymbol{w}_{in} \leftarrow \mu \cdot \boldsymbol{c} + (1 - \mu) \cdot \boldsymbol{w}_{in}.$$

The output of the system is changed towards the desired output by adapting the outgoing weights:

$$\boldsymbol{w}_{out} \leftarrow \eta \cdot \boldsymbol{y} + (1 - \eta) \cdot \boldsymbol{w}_{out}.$$

η and μ are predefined learning rates. Note that (in contrast to the methods described in [15]) 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 center of the hyper-square for which they are responsible. Figure 11 depicts the adaptation of the incoming connections in two different scenarios.

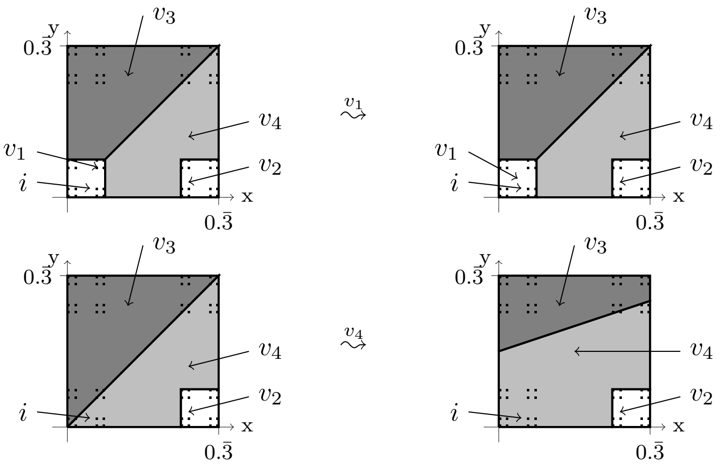


Figure 11. The adaptation of the input weights for a given input i. The first row shows the result of adapting v_1. The second row shows the result if v_1 would not be there and therefore v_4 would be selected as winner. To emphasize the effect, we used a learning rate $\mu = 1.0$, i.e., the winning unit is moved directly into the center of the hyper-square.

Adding new units The adjustment described above enables a certain kind of expansion of the network by allowing units to move to positions where they are responsible for larger areas of the input space. A refinement now should take care of densifying the network in areas where a great error is caused. Every unit will accumulate the error for those training samples, for which it is the winner. If this accumulated error exceeds a given threshold, the unit will be 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 hyper-square containing u. Now u is moved to the center of this area. Information gathered by u during the training process is used to determine a sub-hyper-square into whose center 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 center 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 12.

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 would be removed [15]. If it drops below a threshold, the unit will be removed as depicted in Figure 13.

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 [15], 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 $\mathfrak{C}_0^m$. The preliminary experiments described below show that our method actually works.

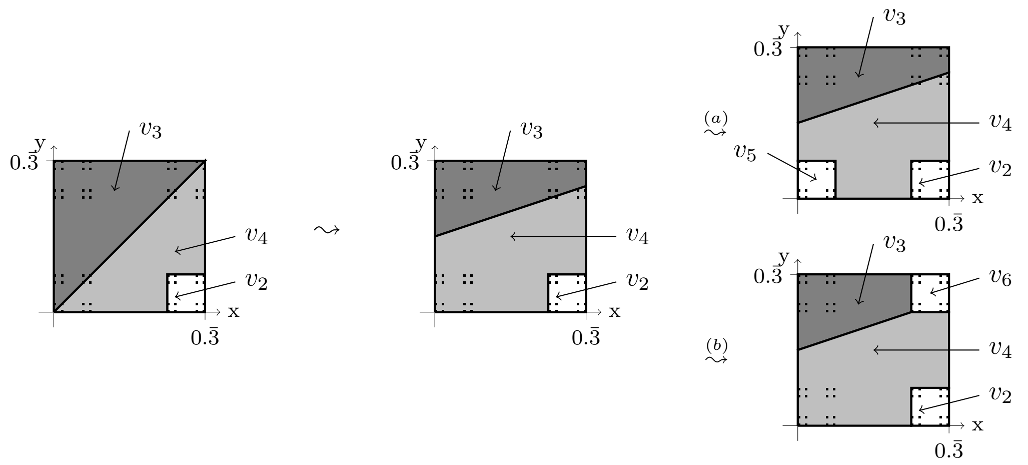


Figure 12. Adding a new unit to support v_4. First, v_4 is moved to the center of the hyper-square it is responsible for. There are four possible sub-hyper-squares to add a new unit. Because v_4 is neither responsible for the north-western, nor for the south-eastern sub-hyper-square, there are two cases left. If most error was caused in the south western sub-hyper-square (a), a new unit (v_5) is added there. If most error was caused in the north-eastern area (b), a new unit (v_6) would be added there.

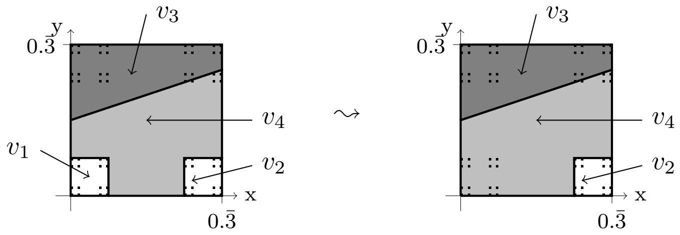


Figure 13. Removing an inutile unit. Let us assume that the outgoing weights of v_1 and v_4 are equal. In this case we would find that the over-all error would not increase if we remove v_1. Therefore its utility would decrease over time until it drops below the threshold and the unit is removed.

6. Evaluation

In this section, we present a proof-of-concept evaluation using our implementation of the first-order core method. The evaluation was set up to show that we indeed achieved the *best of both worlds* with our integration of the neural and the symbolic paradigm. We were thus interested in the capabilities of the integrated system (1) to learn symbolic knowledge, (2) to reason with the acquired knowledge, and (3) to be robust against unit failure. For the experiments, we initialized a network with a wrong program while creating the training samples randomly using the semantic operator of the correct program $\mathcal{P}_2$.

Learning Symbolic Knowledge To illustrate the effects of varying parameters in our system, we used two setups: One with softer utility criteria and one with stricter ones (in the stricter setting the utility decreases faster). Starting from the incorrect initialization, the former decreases the initial error, paying with an increasing number of units, while the latter significantly decreases the number of units, paying with an increasing error.

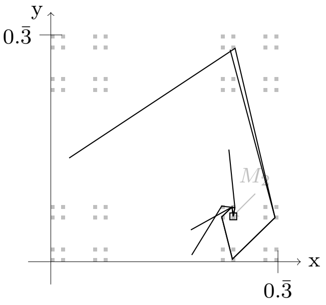


Figure 14. Iterating random inputs. The two dimensions of the input vectors are plotted against each other. The ε-neighborhood of the fixed point M is shown as a small box.

Hence, the performance of the network critically depends on the choice of the parameters. The optimal parameters obviously depend on the concrete setting, e.g., the kind and amount of noise present in the training data, and methods for finding them remain to be investigated. For our experiments we used values which resulted from a mixture of intuition and (non-exhaustive) comparative simulations.

In order to see how suitable our system is for the learning of first-order logic programs, we pitted it against Supervised Growing Neural Gas [15]. Our system clearly outperformed it, which shows that our specialized architecture is superior to the generic architecture in the specific setting for which it was developed.

Reasoning One of the original aims of the core method is to obtain connectionist systems for logic programs which, when iteratively feeding their output back as input, settle to a stable state corresponding to an approximation of a fixed point of the program's single-step operator. In our running example, a unique fixed point is known to exist. To check whether our system reflects this, we proceed as follows: (i) Train a network from scratch until the relative error caused by the network is below 1, i.e., the network outputs are in the ε-neighborhood of the desired outputs. (ii) Transform the obtained network into a recurrent one by connecting the outputs to the corresponding inputs. (iii) Choose a random input vector $\in \mathfrak{C}_0^m$ (which is not necessarily a valid embedded interpretation) and use it as initial input to the network. (iv) Iterate the network until it reaches a stable state, i.e., until the outputs stay inside an ε-neighborhood.

For our example program, the unique fixed point of $T_{\mathcal{P}_2}$ is M_2 as given in Example 4. Figure 14 shows the input space and the ε-neighborhood of M_2, along with all intermediate results of the iteration for 5 random initial inputs. The example computations converge, because the underlying program is acyclic [22,19]. After at most 6 steps, the network is stable in all cases in the sense that all outputs stay within the ε-neighborhood. This coincides with the number of applications of the $T_{\mathcal{P}_2}$ operator required to fix the significant atoms, which confirms that the training method really implements our intention of learning $T_{\mathcal{P}_2}$. The fact that even a network obtained through training from scratch converges in this sense further underlines the efficiency of our training method.

The experiment shows that the network acquired a representation of the target program in such a way that it can be used to compute the desired model of the logic program. Since this represents the conclusions which can be drawn by logical deduction, we can conclude that our connectionist system is indeed able to reason with the encoded knowledge.

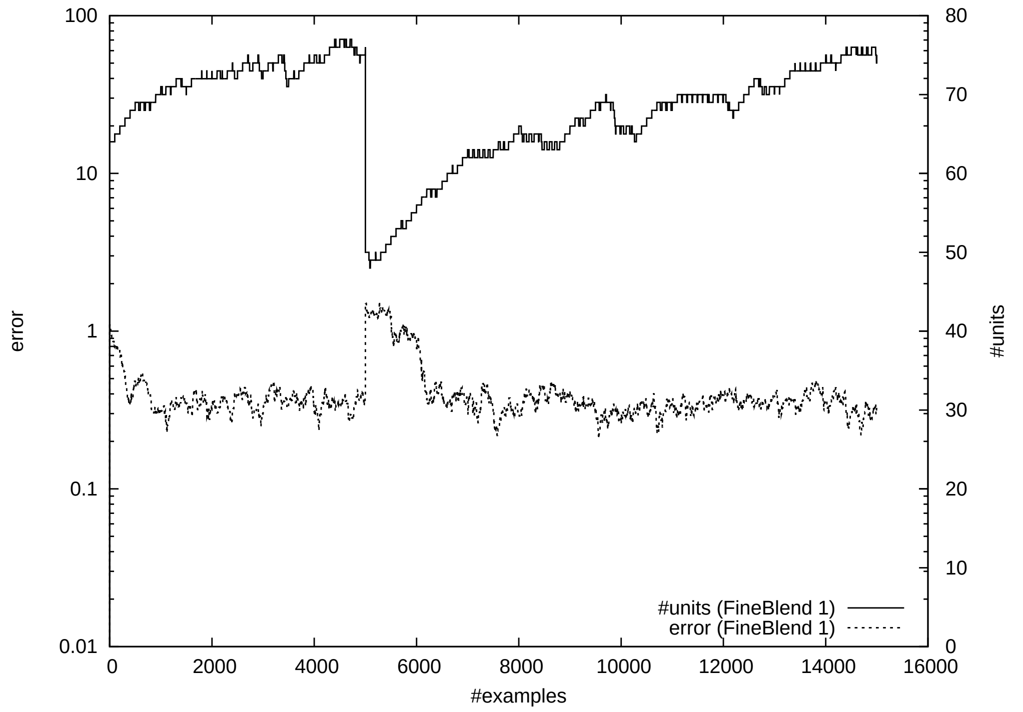


Figure 15. The effects of unit failure. We use a logarithmic scale for the error axis, and the error values are relative to ε, i.e., a value of 1 designates an absolute error of ε.

Robustness The described system is able to handle noisy data and to cope with damage. Indeed, the effects of damage to the system are quite obvious: If a hidden unit u fails, the receptive area is taken over by other units, thus only the specific results learned for u's receptive area are lost. While a corruption of the input weights may cause no changes at all in the network function, generally it can alter the unit's receptive area. If the output weights are corrupted, only certain inputs are effected. If the damage to the system occurs during training, it will be repaired very quickly as indicated by the following experiment. Noise is generally handled gracefully, because wrong or unnecessary adjustments or refinements can be undone in the further training process.

To demonstrate the effects of unit failure, a FineBlend 1 network is initialized and refined through training with 5000 samples, then one third of its hidden units are removed randomly, and the training is continued as if nothing had happened. Figure 15 shows that the network handles the damage gracefully and recovers quickly. The error increases only slightly and drops back very soon; the number of units continues to increase to the previous level, recreating the redundancy necessary for robustness.

While it is no surprise that a connectionist system is robust against unit failure, we would like to emphasize that the trained network actually encodes symbolic knowledge, and knowledge bases are generally *fragile* against loss of information. Robustness in conjunction with the reasoning abilities reported earlier is, in our opinion, very remarkable.

7. Further Work

While our work shows that it is possible to build integrated neural-symbolic systems for first-order logic which combine the *best of both worlds*, much remains to be done to realize applications based on it. Obviously, it will be necessary to further evaluate our system and to test it using larger and real world examples. This will also necessitate a reimplementation suitable for such a task. It would certainly also be of interest to compare our system with others, although such a comparison will only be partial due to the limitations which other neural-symbolic approaches currently have (see Section 8). We expect that such tests together with an in-depth analysis of the system will also provide heuristics to determine optimal values for the parameters of the system.

Further work will also be required in order to take full advantage of the symbolic aspect of our integrated system. More precisely, it shall be necessary to complete the neural-symbolic cycle by developing algorithms which are able to extract knowledge from the trained network in logical form. We note, however, that while the extraction of propositional rules from trained networks is reasonable well understood, the extraction of first-order rules is still a very open issue, and entirely new ideas will be needed to address it. A starting point may be to interpret a (trained) neural network N as an x-dimensional real vector consisting of the x parameters which define the network (i.e., the weights). A search space is now spanned by a set of first-order logic programs. Each $\mathcal{P}$ in this set can be cast into a network $\mathcal{N}_{\mathcal{P}}$ using our representation algorithm. It should then be possible to select the $\mathcal{P}$ for which $\mathcal{N}_{\mathcal{P}}$ is *closest* to $\mathcal{N}$ in $\mathbb{R}^x$, and this $\mathcal{P}$ could be interpreted as the *extraction* of $\mathcal{N}$. Obviously, a number of obstacles need to be resolved before this idea can be realized, including the questions about the appropriate way of measuring *distance* between two networks in $\mathbb{R}^x$. Further insights from fractal geometry could also lead to a novel extraction method. In particular ideas from fractal image encoding [8] together with the results from [3] seem to be promising. But all this remains to be done.

8. Related Work

In Section 3, we already mentioned some work related to the core method. Literature reporting on various attempts and methods for the connectionist dealing with structured knowledge exists in abundance. Therefore, we will refer to the survey article [4] and to the book [18] only, containing state of the art contributions by leading researchers.

Concerning the more specific problem of dealing with first-order predicate logic, there are only few other approaches worth mentioning. Most famous is the SHRUTI system [33,32] which is a connectionist system which is able to do reasoning over first-order logic in a parallelized way. SHRUTI, however, has only very limited learning capabilities, and can process only a very limited fragment of first-order logic in a sound and complete way. An entirely different approach is taken in [17], where first-order logical formulae are expressed in variable-free form by means of topos theory, and this is taken as a basis for a connectionist system which is able to learn models of first-order theories. [25], finally, is based on the propositional core method. The underlying idea is to first approximate a target first-order logic program by a (finite) propositional logic program and then use the propositional core method for encoding this program. The work reported in [25] is purely theoretical in nature, and it is unclear whether it can lead to a useful implementation.

9. Conclusion

We presented the first integrated neural-symbolic system with the following features:

(i) It can learn knowledge in the form of first-order logic programs.
(ii) It outperforms other less specialized approaches in the learning task.
(iii) It can be used for reasoning over the learned knowledge in the form of model generation for the encoded logic program.
(iv) It is robust against unit failure.

While our results are preliminary in the sense that much work remains to be done to realize a practically applicable system, our work shows that the *propositional fixation* of many neural-symbolic systems can be overcome.

Acknowledgments: Many thanks go to Sven-Erik Bornscheuer, Artur d'Avila Garcez, Yvonne McIntyre (formerly Kalinke), Anthony K. Seda, Hans-Peter Störr, Andreas Witzel and Jörg Wunderlich, who all contributed to the core method.

References

[1] 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.

[2] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[3] S. Bader and P. Hitzler. Logic programs, iterated function systems, and recurrent radial basis function networks. 2(3):273–300, 2004. Special Issue on Neural-Symbolic Systems.

[4] S. Bader and P. Hitzler. Dimensions of neural-symbolic integration – a structured survey. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, pages 167–194. King's College Publications, 2005.

[5] S. Bader, P. Hitzler, S. Hölldobler, and A. 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.

[6] S. Bader and S. Hölldobler. The Core Method: Connectionist model generation. In *Proceedings of the ICANN'06*, volume 4132 of *Lecture Notes in Computer Science*, pages 1–13, 2006.

[7] D. H. Ballard. Parallel logic inference and energy minimization. pages 203–208, 1986.

[8] M. Barnsley. *Fractals Everywhere.* Academic Press, San Diego, CA, USA, 1993.

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

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

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

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

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

[14] M. Fitting. Metric methods, three examples and a theorem. *Journal of Logic Programming*, 21(3):113–127, 1994.

[15] B. Fritzke. *Vektorbasierte Neuronale Netze.* Habilitation, Technische Universität Dresden, 1998.

[16] K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. 2:183–192, 1989.

[17] H. Gust, K-U. Kühnberger, and P. Geibel. Learning models of predicate logical theories with neural networks based on topos theory. In Hammer and Hitzler [18], pages 233–264.

[18] B. Hammer and P. Hitzler, editors. *Perspectives of Neural-Symbolic Integration*, volume 77 of *Studies in Computational Intelligence.* Springer, Berlin, 2007.

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

[20] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. 2:359–366, 1989.

[21] S. Hölldobler and Y. 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.

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

[23] 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.

[24] Y. Kalinke. Ein massiv paralleles Berechnungsmodell für normale logische Programme. Master's thesis, TU Dresden, Fakultät Informatik, 1994. (in German).

[25] M. Lane and A.K. Seda. Some aspects of the integration of connectionist and logic-based systems. *Information*, 9(4):551–562, 2006.

[26] T. E. Lange and M. G. Dyer. High-level inferencing in a connectionist network. *Connection Science*, 1:181–217, 1989.

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

[28] J. McCarthy. Epistemological challanges for connectionism. *Behavioural and Brain Sciences*, 11:44, 1988. Commentary to [34].

[29] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[30] G. Pinkas. Symmetric neural networks and logic satisfiability. *Neural Computation*, 3:282–291, 1991.

[31] A.K. Seda. Topology and the semantics of logic programs. *Fundamenta Informaticae*, 24(4):359–386, 1995.

[32] L. Shastri. SHRUTI: A neurally motivated architecture for rapid, scalable inference. In Hammer and Hitzler [18], pages 183–204.

[33] L. Shastri and V. 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.

[34] P. Smolensky. On variable binding and the representation of symbolic structures in connectionist systems. Technical Report CU-CS-355-87, Department of Computer Science & Institute of Cognitive Science, University of Colorado, Boulder, CO 80309-0430, 1987.

[35] G.G. Towell and J.W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, 1993.

[36] S. Willard. *General Topology.* Addison–Wesley, 1970.