# 9

# The Core Method: Connectionist Model Generation for First-Order Logic Programs

Sebastian Bader,[1*] Pascal Hitzler[2], Steffen Hölldobler[1] and Andreas Witzel[3†]

[1] International Center for Computational Logic
Technische Universität Dresden
01062 Dresden, Germany
Sebastian.Bader@inf.tu-dresden.de, sh@iccl.tu-dresden.de
[2] Institute AIFB
Universität Karlsruhe
76131 Karlsruhe, Germany
hitzler@aifb.uni-karlsruhe.de
[3] Institute for Logic, Language and Computation
Universiteit van Amsterdam
1018 TV Amsterdam, The Netherlands
awitzel@illc.uva.nl

**Summary.** Research into the processing of symbolic knowledge by means of connectionist networks aims at systems which combine the declarative nature of logic-based artificial intelligence with the robustness and trainability of artificial neural networks. This endeavour has been addressed quite successfully in the past for propositional knowledge representation and reasoning tasks. However, as soon as these tasks are extended beyond propositional logic, it is not obvious at all what neural-symbolic systems should look like such that they are truly connectionist and allow for a declarative reading at the same time.

The Core Method – which we present here – aims at such an integration. It is a method for connectionist model generation using recurrent networks with feed-forward core. These networks can be trained by standard algorithms to learn symbolic knowledge, and they can be used for reasoning about this knowledge.

## 9.1 Introduction

From the very beginning, artificial neural networks have been related to propositional logic. McCulloch-Pitts networks, as presented in the seminal paper

* Sebastian Bader was supported by the GK334 of the German Research Foundation.

† Andreas Witzel was supported by a Marie Curie Early Stage Research fellowship in the project GloRiClass (MEST-CT-2005-020841).

from 1943 [1], represent propositional formulae. Finding a global minimum of the energy function modeling a symmetric network corresponds to finding a model of a propositional logic formula and vice versa [2]. These are just two examples for the strong relation between neural networks and propositional logic, which is well-known and well-studied. However, similar research concerning more expressive logics did not start until the late 1980s, which prompted John McCarthy to talk about the *propositional fixation* of connectionist systems in [3].

Since then, there have been numerous attempts to model first-order fragments in connectionist systems.[4] In [5] energy minimization was used to model inference processes involving unary relations. In [6] and [7] multi-place predicates and rules over such predicates are modeled. In [8] 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 their capabilities were limited at best. Recursive auto-associative memories based on ideas first presented in [9], holographic reduced representations [10] or the networks used in [11] have considerable problems with deeply nested structures. We are unaware of any connectionist system that fully incorporates the power of symbolic knowledge and computation as argued for in e.g. [12], and indeed there remain many open challenges on the way to realising this vision [13].

In this chapter we are mainly interested in knowledge based artificial neural networks, i.e., networks which are initialized by available background knowledge before training methods are applied. In [14] it has been shown that such networks perform better than purely empirical and hand-built classifiers. [14] uses background knowledge in the form of propositional rules and encodes these rules in multi-layer feed-forward networks. Independently, we have developed a connectionist system for computing the least model of propositional logic programs if such a model exists [15]. This system has been further developed to the so-called *Core Method*: background knowledge represented as logic programs is encoded in a feed-forward network, recurrent connections allow for a computation or approximation of the least model of the logic program (if it exists), training methods can be applied to the feed-forward kernel in order to improve the performance of the network, and, finally, an improved program can be extracted from the trained kernel closing the neural-symbolic cycle as depicted in Figure 9.1.

In this chapter, we will present the Core Method in Section 9.3. In particular, we will discuss its propositional version including its relation to [14] and its extensions. The main focus of this paper will be on extending the Core Method to deal with first-order logic programs in Section 9.4. In particular, we will give a feasibility result, present a first practical implementation, and discuss preliminary experimental data in Section 9.5. These main sections

[4] See also the survey [4].

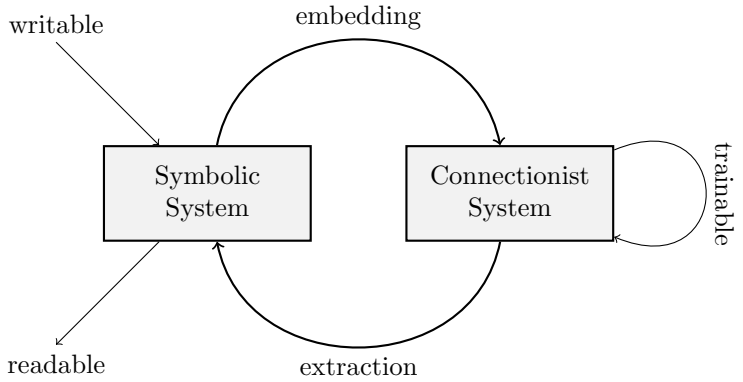


**Fig. 9.1.** The Neural-Symbolic Cycle.

are framed by introducing basic notions and notations in Section 9.2 and an outlook in Section 9.6.

## 9.2 Preliminaries

We assume the reader to be familiar with basic notions from artificial neural networks and logic programs and refer to e.g. [16] and [17], respectively. Nevertheless, we repeat some basic notions.

### 9.2.1 Logic Programs

A *logic program* over some first order 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 $L_i$ 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 1.* The following propositional logic program will serve as our running example in Section 9.3.

$$
\begin{aligned}
\mathcal{P}_1 = \{\ & p, && \%\ p \text{ is always true.}\\
& r \leftarrow p \wedge \neg q, && \%\ r \text{ is true if } p \text{ is true and } q \text{ is false.}\\
& r \leftarrow \neg p \wedge q\ \} && \%\ r \text{ is true if } p \text{ is false and } q \text{ is true.}
\end{aligned}
$$

*Example 2.* The following (first order) logic program will serve as our running example in Section 9.4.

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

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 *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 logic program $P$ is called *acyclic* if there exists a level mapping $|\cdot|$ such that for all clauses $A \leftarrow L_1 \wedge \cdots \wedge L_n \in \mathcal{G}(P)$ we have $|A| > |L_i|$ for $1 \leq i \leq n$.

*Example 3.* Consider the program from Example 2 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:

$$\begin{aligned}|\cdot| : \mathcal{B}_\mathcal{L} &\to \mathbb{N}^+\\ e(s^n(0)) &\mapsto 2n+1\\ o(s^n(0)) &\mapsto 2n+2.\end{aligned}$$

A *(Herbrand) interpretation* $I$ is a subset of $\mathcal{B}_\mathcal{L}$. Those atoms $A$ with $A \in I$ are said to be *true* under $I$, those with $A \notin I$ 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 the program $\mathcal{P}_2$ from Example 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* $T_\mathcal{P} : \mathcal{I}_\mathcal{L} \to \mathcal{I}_\mathcal{L}$ 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* of the program [18]. In Example 2, the (obviously intended) model $M_2$ is supported, while $\mathcal{B}_\mathcal{L}$ is a model but not supported.

### 9.2.2 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, respectively. The gray shaded area in Figure 9.2 shows a simple feed-forward network. In this paper we will construct recurrent networks by connecting the output units of a feed-forward network $N$ to the input units of $N$. Figure 9.2 shows a blueprint of such a recurrent network.

## 9.3 The Core Method for Propositional Logic

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 leading to a stable state, corresponding to the least model of the logic program provided that such a least model exists (see Figure 9.2). Moreover, the core can be trained using standard methods from connectionist systems. In other words, we are considering connectionist model generation using recurrent networks with feed-forward core.

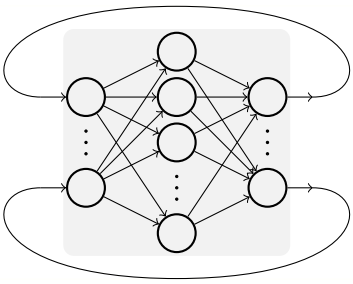

**Fig. 9.2.** The blueprint of a recurrent network used by the Core Method.

The ideas behind the Core Method for propositional logic programs were first presented in [15] (see also [19]). Consider the logic program from Example 1. A translation algorithm turns such a 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. Such interpretations 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 logical threshold 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; moreover this unit activates the output layer unit corresponding to the postcondition of the rule. Figure 9.3 shows the network obtained by the translation algorithm if applied to $\mathcal{P}_1$.

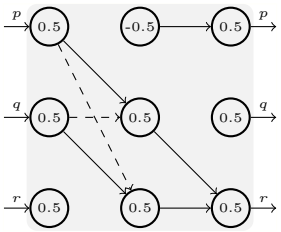


**Fig. 9.3.** The core corresponding to $\mathcal{P}_1 = \{p,\ r \leftarrow p \wedge \neg q,\ r \leftarrow \neg p \wedge q\}$ . Solid connections have weight 1.0, dashed connections have weight $-1.0$. The numbers within the units denote the thresholds.

In [15] 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 and yields the unique fixed point of $T_{\mathcal{P}}$.

The use of logical threshold units in [15] made it easy to prove these results. However, it prevented the application of standard training methods like backpropagation to the kernel. This problem was solved in [20] by showing that the same results can be achieved if bipolar sigmoidal units are used instead (see also [21]). [20] also overcomes a restriction of the KBANN method originally presented in [14]: 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 [22] three-valued logic programs are discussed; This approach has been extended in [23] (see also the chapter by Komendantskaya, Lane and Seda in this volume) to finitely determined sets of truth values. Modal logic programs have been considered in [24] (see also the chapter by Garcez in this volume). Answer set programming and meta-level priorities are discussed in [21]. The Core Method has been applied to intuitionistic logic programs in [25].

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 process is outside the scope of this chapter and interested readers are referred to e.g. [26] or [21].

## 9.4 The Core Method and First-Order Logic

First- or higher-order logics are the primary choice if structured objects and structure-sensitive processes are to be modeled in Artificial Intelligence. In particular, first-order logic plays a prominent role because any computable function can be expressed by first-order logic 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.

We will first show in Section 9.4.1 that an extension of the core method to first-order logic programs is feasible. However, the result will be purely theoretical and thus the question remains how core-networks can be constructed for first-order programs. In Section 9.4.2, a first practical solution is discussed, which approximates the meaning functions of logic programs by means of piecewise constant functions. This approach is extended to a multi-dimensional setting in Section 9.4.3 allowing for arbitrary precision, even if implemented on a real computer. A novel training method, tailored for our specific setting, is discussed in Section 9.4.4. Some preliminary experimental data are presented in Section 9.5.

### 9.4.1 Feasibility

It is well known that multilayer feed-forward networks are universal approximators [27, 28] for certain functions of the type $\mathbb{R}^n \to \mathbb{R}^m$. Hence, if we find a 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 functions of such programs.

As proposed in [29], we use level mappings to bridge the gap between the space of interpretations and the real numbers.

**Definition 1.** *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 $\iota$ as follows:*

$$\begin{aligned} \iota : \mathcal{I}_\mathcal{L} &\to \mathbb{R} \\ I &\mapsto \sum_{A \in I} b^{-|A|}. \end{aligned}$$

*We will use* $\mathfrak{C}$ *to denote the set of all embedded interpretations:*

$$\mathfrak{C} := \{\iota(I) \mid I \in \mathcal{I}_\mathcal{L}\} \subset \mathbb{R}$$

For $b > 2$, we find that $\iota$ is an injective mapping from $\mathcal{I}_\mathcal{L}$ to $\mathbb{R}$ and a bijection between $\mathcal{I}_\mathcal{L}$ and $\mathfrak{C}$.[5] Hence, we have a sound and complete encoding of interpretations.

**Definition 2.** *Let* $\mathcal{P}$ *be a logic program and* $T_\mathcal{P}$ *its associated meaning operator. We define a sound and complete encoding* $f_P : \mathfrak{C} \to \mathfrak{C}$ *of* $T_\mathcal{P}$ *as follows:*

$$f_P(r) = \iota(T_\mathcal{P}(\iota^{-1}(r))).$$

In [29] we proved – among other results – that for each logic program $\mathcal{P}$ which is acyclic wrt. a bijective level mapping the function $f_P$ is contractive,[6] hence continuous. Moreover, $\mathfrak{C}$ 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 [28]. This shows that the meaning function of a logic program (of the kind discussed before) can be approximated by a core. (ii) Considering an appropriate metric, which will be discussed in a moment, we can apply Banach's contraction mapping theorem (see e.g. [30]) 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 [29] 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.

But what exactly is the approximation of an interpretation or a model in this context? Let $\mathcal{P}$ be a logic program and $l$ a level mapping. We can define a metric $d$ on interpretations $I$ and $J$ as follows:

$$d(I,J) = \begin{cases} 0 & \text{if } I = J, \\ 2^{-n} & \text{if } n \text{ is the smallest level on which } I \text{ and } J \text{ disagree.} \end{cases}$$

As shown in [31] the set of all interpretations together with $d$ is a complete metric space. Moreover, an interpretation $I$ *approximates* an interpretation $J$ *to degree* $n \in \mathbb{N}$ iff $d(I,J) \leq 2^{-n}$. In other words, if a recurrent network approximates the least model $I$ of an acyclic 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 $I(A) = \iota^{-1}(r)(A)$.

**Theorem 1.** *Let* $\mathcal{P}$ *be an acyclic logic program with respect to some bijective level mapping* $\iota$*. Then there exists a 3-layer core-network with sigmoidal activation function in the hidden layer approximating* $T_\mathcal{P}$ *up to a given accuracy* $\varepsilon > 0$*.*

[5] For $b = 2$, $\iota$ is not injective, as $0.0\bar{1}_2 = 0.1_2$.

[6] This holds for $b > 3$ only. Therefore, we will use $b = 4$ throughout this article.

*Proof (sketch).* This theorem follows directly from the following facts:

1. $T_{\mathcal{P}}$ for an acyclic logic program is contractive [31].
2. $f_P$ is a sound and complete embedding of $T_{\mathcal{P}}$ into the real numbers which is contractive and hence continuous for acyclic logic programs [29].
3. $\mathfrak{C} \subset \mathbb{R}$ is compact [29].
4. Every continuous function on a compact subset of $\mathbb{R}$ can be uniformly approximated by feed-forward networks with sigmoidal units in the hidden layer (Funahashi's theorem [28]).

Theorem 1 shows the existence of an approximating core network. Similar results for more general programs can also be obtained and are reported in [19]. Moreover, in [29] it is also shown that a recurrent network with a core network as stated in Theorem 1 approximates the least fixed point of $T_{\mathcal{P}}$ for an acyclic logic program $\mathcal{P}$. But as mentioned above, these results are purely theoretical. Networks are known to exist, but we do not yet know how to construct them given a logic program. We will address this next.

### 9.4.2 Embedding

In this section, we will show how to construct a core network approximating the meaning operator of a given logic program. As above, we will consider logic programs $\mathcal{P}$ which are acyclic wrt. a bijective level mapping. We will construct sigmoidal networks and RBF networks with a raised cosine activation function. All ideas presented here can be found in detail in [32]. To illustrate the ideas, we will use the program $\mathcal{P}_2$ given in Example 2 as a running example. The construction consists of five steps:

1. Construct $f_P$ as a real embedding of $T_{\mathcal{P}}$.
2. Approximate $f_P$ using a piecewise constant function $\bar{f}_P$.
3. Implement $\bar{f}_P$ using (a) step and (b) triangular functions.
4. Implement $\bar{f}_P$ using (a) sigmoidal and (b) raised cosine functions.
5. Construct the core network approximating $f_P$.

In the sequel we will describe the ideas underlying the construction. A rigorous development including all proofs can be found in [32, 33]. One should observe that $f_P$ 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 concerned with accuracy on $\mathfrak{C}$ only.

#### Construct $f_P$ as a real embedding of $T_{\mathcal{P}}$:

We use $f_P(r) = \iota(T_{\mathcal{P}}(\iota^{-1}(r)))$ as a real-valued version for $T_{\mathcal{P}}$. As mentioned above, $f_P$ is a sound and complete encoding of the immediate consequence operator. 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 [34]. 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 $\iota$. 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. This process is repeated ad infinitum and we find $\mathfrak{C}$ to be its limit, i.e. it is the intersection of all $\mathfrak{C}_n$, where $\mathfrak{C}_n$ denotes the result after $n$ splits and removals. The first four iterations (i.e. $\mathfrak{C}_0$, $\mathfrak{C}_1$, $\mathfrak{C}_2$ and $\mathfrak{C}_3$) are depicted in Figure 9.4.

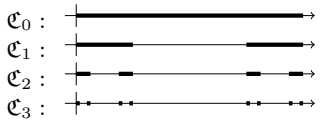


**Fig. 9.4.** The first four iterations ($\mathfrak{C}_0$, $\mathfrak{C}_1$, $\mathfrak{C}_2$ and $\mathfrak{C}_3$) of the construction of $\mathfrak{C}$ for $b = 4$.

*Example 5.* Considering program $\mathcal{P}_2$, and the level mapping from Example 3 ($|e(s^n(0))| := 2n + 1$, $|o(s^n(0))| := 2n + 2$), we obtain $f_{\mathcal{P}_2}$ as depicted in Figure 9.5 on the left.

**Approximate $f_P$ using a piecewise constant function $\bar{f}_P$:**

Under the assumption that $\mathcal{P}$ is acyclic, we find that all variables occurring in the precondition of a rule are also contained in its postcondition. 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 $T_{\bar{\mathcal{P}}} = \bar{T}_{\mathcal{P}}$. We will use $\bar{f}_P$ to denote the embedding of $\bar{T}_{\mathcal{P}}$ and we find that $\bar{f}_P := f_{\bar{\mathcal{P}}}$ is a piecewise constant function, being constant on each connected interval of $\mathfrak{C}_{n-1}$. Furthermore, we find that $|f_P(x) - \bar{f}_P(x)| \leq 2^{-n}$ for all $x \in \mathfrak{C}$, i.e., $\bar{f}_P$ is a constant piecewise approximation of $f_P$.

*Example 6.* For our running example $\mathcal{P}_2$ and $n = 3$, we obtain

$$\begin{aligned}\bar{\mathcal{P}}_2 = \{\ & e(0).\\ & e(s(0)) \leftarrow o(0).\\ & o(0) \leftarrow \neg e(0).\quad \}\end{aligned}$$

and $\bar{f}_{\mathcal{P}_2}$ as depicted in Figure 9.5 on the right.

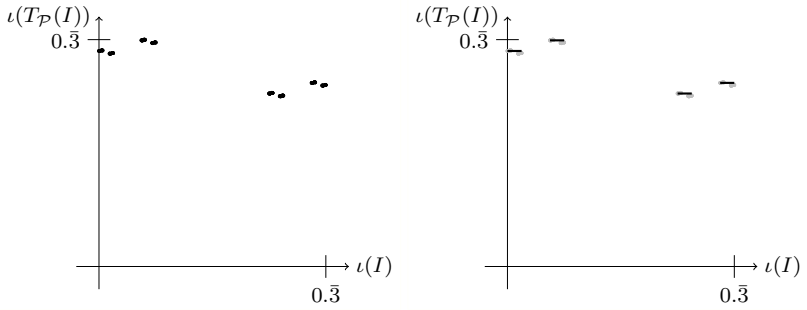


**Fig. 9.5.** On the left is the plot of $f_{\mathcal{P}_2}$. On the right a piecewise constant approximation $\bar{f}_{\mathcal{P}_2}$ (for level $n = 3$) of $f_{\mathcal{P}_2}$ (depicted in light gray) is shown.

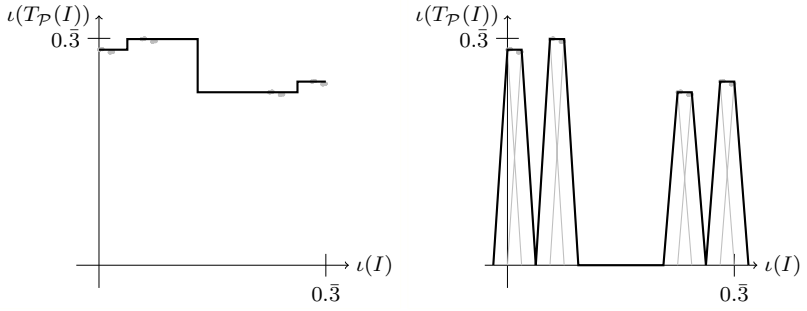


**Fig. 9.6.** Two linear approximations of $\bar{f}_{\mathcal{P}_2}$. On the left, three step functions were used; On the right, eight triangular functions (depicted in gray) add up to the approximation, which is shown using thick lines.

### Implement $\bar{f}_P$ using (a) step and (b) triangular functions:

As a next step, we will show how to implement $\bar{f}_P$ using step and triangular functions. Those functions are the linear counterparts of the functions actually used in the networks constructed below. If $\bar{f}_P$ 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 9.6 on the left.

Each constant piece of length $\lambda := \frac{1}{b-1} \cdot \frac{1}{b^n}$ could also be implemented using two triangular functions with width $\lambda$ which are centered at the endpoints. Those two triangles 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. The triangular implementation is depicted in Figure 9.6 on the right.

### Implement $\bar{f}_P$ 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 described above 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.

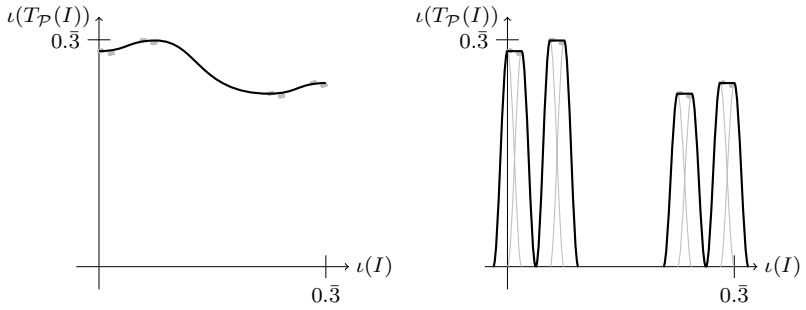


**Fig. 9.7.** Two non-linear approximations of $\bar{f}_{\mathcal{P}_2}$. On the left, sigmoidal functions were used and on the right, raised cosines.

**Construct the core network approximating $f_P$:**

A standard sigmoidal core network approximating the $T_\mathcal{P}$-operator of a given program $\mathcal{P}$ 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, one for each raised cosine functions. Detailed constructions can be found in [32].

A constructive proof for Theorem 1 is now possible. It follows directly from the fact that the network constructed as above approximates a given $T_\mathcal{P}$-operator up to any given accuracy. As the proof of the correctness of the

construction is rather straightforward but tedious, we omit it here and refer the interested reader to [32, 33].

In this first approach we used a one-dimensional embedding to obtain a unique real number $\iota(I)$ for a given interpretation $I$. Unfortunately, the precision of a real computer is limited, which implies that using e.g. a 32-bit computer we could embed the first 16 atoms only. We address this problem in the next section.

### 9.4.3 Multi-Dimensional Embedding

We have just seen 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. For various reasons, spelled out in [35, 36], we will use an RBF network approach, inspired by vector-based networks as described in [37]. Analogously to the previous section, we will proceed as follows:

1. Construct $f_P$ as a real embedding of $T_\mathcal{P}$.
2. Approximate $f_P$ using a piecewise constant functions $\bar{f}_P$.
3. Construct the core network approximating $f_P$.

I.e., 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.

Additionally, we will present a novel training method, tailored for our specific setting. The system presented here is a fine blend of techniques from the *Supervised Growing Neural Gas (SGNG)* [37] and our embedding.

#### Construct $f_P$ 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 multi-dimensional embedding of $T_\mathcal{P}$.

**Definition 3.** *An* $m$-dimensional level mapping *is a bijective function* $\|\cdot\| : \mathcal{B}_\mathcal{L} \to (\mathbb{N}^+, \{1, \ldots, m\})$. *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 7.* Reconsider the program from Example 2. A possible 2-dimensional level mapping is given as:

$$\begin{aligned} \|\cdot\| : \mathcal{B}_\mathcal{L} &\to (\mathbb{N}^+, \{1,2\}) \\ e(s^n(0)) &\mapsto (n+1, 1) \\ o(s^n(0)) &\mapsto (n+1, 2). \end{aligned}$$

I.e., all *even*-atoms are mapped to the first dimension, whereas the *odd*-atoms are mapped to the second.

**Definition 4.** *Let $b \geq 3$ and let $A \in \mathcal{B}_\mathcal{L}$ be an atom with $\|A\| = (l, d)$. The* $m$-dimensional embedding $\boldsymbol{\iota} : \mathcal{B}_\mathcal{L} \to \mathbb{R}^m$ *is defined as:*

$$\boldsymbol{\iota}(A) := (\iota_1(A), \ldots, \iota_m(A)) \quad \text{with} \quad \iota_j(A) := \begin{cases} b^{-l} & \text{if } j = d \\ 0 & \text{otherwise} \end{cases}$$

*The extension $\boldsymbol{\iota} : \mathcal{I}_\mathcal{L} \to \mathbb{R}^m$ is obtained as:*

$$\boldsymbol{\iota}(I) := \sum_{A \in I} \boldsymbol{\iota}(A).$$

*We will use $\mathfrak{C}^m$ to denote the set of all embedded interpretations:*

$$\mathfrak{C}^m := \{\boldsymbol{\iota}(I) \mid I \in \mathcal{I}_\mathcal{L}\} \subset \mathbb{R}^m.$$

As mentioned above, $\mathfrak{C}^1$ is the classical Cantor set and $\mathfrak{C}^2$ the 2-dimensional variant of it [34]. Obviously, $\boldsymbol{\iota}$ is injective for a bijective level mapping and it is bijective on $\mathfrak{C}^m$. Without going into detail, Figure 9.8 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 9.8 depicts $\mathfrak{C}^2_0, \mathfrak{C}^2_1, \mathfrak{C}^2_2$ and $\mathfrak{C}^2_3$. For readers with background in fractal geometry we note, that these are the first 4 applications of an appropriately set up iterated function system [34].

*Example 8.* Using the 1-dimensional level mapping from Example 3, we obtain $\mathfrak{C}^1$ as depicted in Figure 9.8 at the top. Using the 2-dimensional from above, we obtain $\mathfrak{C}^2$ and $\boldsymbol{\iota}(M_2) = (0.10\overline{10}_4, 0.01\overline{01}_4) \approx (0.2666667, 0.0666667)$ for the embedding of the intended model $M_2$.

Using the $m$-dimensional embedding, the $T_\mathcal{P}$-operator can be embedded into the real vectors to obtain a real-valued function $f_P$.

**Definition 5.** *Let $\boldsymbol{\iota}$ be an $m$-dimensional embedding as introduced in Definition 4. The* $m$-dimensional embedding $f_P$ *of a given $T_\mathcal{P}$-operator is defined as:*

$$\begin{aligned} f_P : \mathfrak{C}^m &\to \mathfrak{C}^m \\ \mathbf{x} &\mapsto \boldsymbol{\iota}\left(T_\mathcal{P}\left(\boldsymbol{\iota}^{-1}(\mathbf{x})\right)\right). \end{aligned}$$

This embedding of $T_\mathcal{P}$ is preferable to the one presented above, because it allows for better approximation precision on real computers.

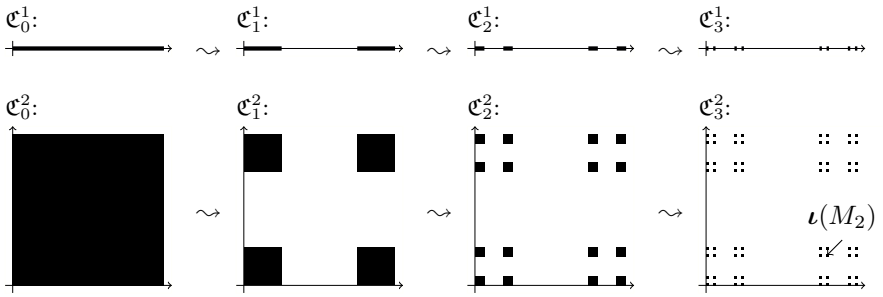


**Fig. 9.8.** The first four iterations of the construction of $\mathfrak{C}^1$ are depicted on the top. The construction of $\mathfrak{C}^2$ is depicted below containing $\boldsymbol{\iota}(M_2)$ from Example 8. Both are constructed using $b = 4$.

### Approximate $f_P$ using a piecewise constant function $\bar{f}_P$:

As above and under the assumption that $\mathcal{P}$ is acyclic, we can approximate $T_\mathcal{P}$ up to some level $n$ by some $\bar{T}_\mathcal{P}$. After embedding $\bar{T}_\mathcal{P}$ into $\mathbb{R}^m$, we find that it is constant on certain regions, namely on all connected intervals in $\mathfrak{C}^m_{n-1}$. 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 $\mathfrak{C}^m_l$. An approximation of $T_\mathcal{P}$ up to some level $n$ will yield a function which is constant on all hyper-squares of level $n-1$.

*Example 9.* Considering program $\mathcal{P}_2$, and the level mapping from Example 7, we obtain $\bar{f}_P$ for $n = 3$ as depicted in Figure 9.9.

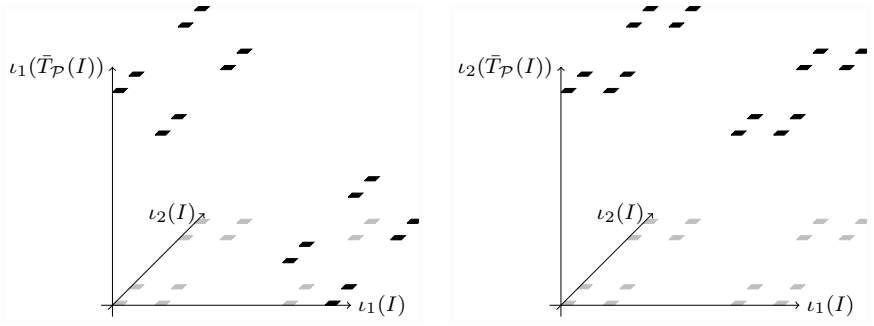


**Fig. 9.9.** $\bar{f}_P$ for $\mathcal{P}_2$, the 2-dimensional level mapping from Example 7 and $n = 3$. The outputs for dimension 1 and 2 are shown on the left and on the right, respectively.

To simplify the notations later, we introduce the *largest exclusive hyper-square* and the *smallest inclusive hyper-square* as follows.

**Definition 6.** *The* largest exclusive hyper-square *of a vector* $\mathbf{u} \in \mathfrak{C}_0^m$ *and a set of vectors* $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_k\} \subseteq \mathfrak{C}_0^m$*, denoted by* $H_{ex}(\mathbf{u}, V)$*, either does not exist or is the hyper-square* $H$ *of least level for which* $\mathbf{u} \in H$ *and* $V \cap H = \emptyset$*. The* smallest inclusive hyper-square *of a non-empty set of vectors* $U = \{\mathbf{u}_1, \ldots, \mathbf{u}_k\} \subseteq \mathfrak{C}_0^m$*, denoted by* $H_{in}(U)$*, is the hyper-square* $H$ *of greatest level for which* $U \subseteq H$*.*

*Example 10.* 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 9.10. 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 $\mathfrak{C}_0$.

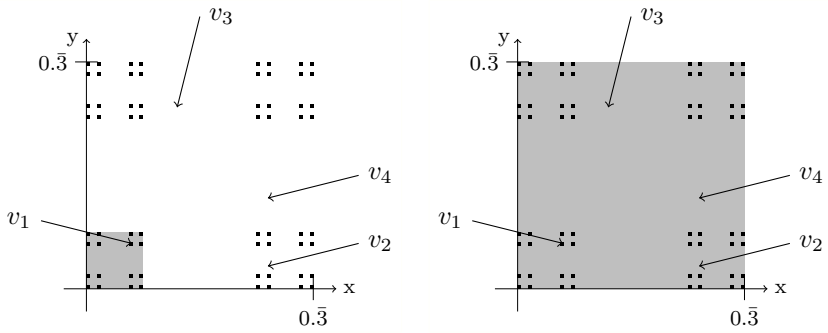


**Fig. 9.10.** The largest exclusive hyper-square of $v_1$ with respect to the set $\{v_2, v_3, v_4\}$ is shown on the left and the smallest inclusive hyper-square of the set $\{v_1, v_2, v_3, v_4\}$ is shown on the right. Both are depicted in light gray together with $\mathfrak{C}^2$ in black.

**Construct the core network approximating $f_P$:**

We will use a 3-layered network with a winner-take-all hidden layer. For each hyper-square $H$ of level $n-1$, 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 $\bar{f}_P$ on that hyper-square. Thus, we obtain a connectionist network approximating the semantic operator $T_\mathcal{P}$ up to the given accuracy $\varepsilon$.

To determine the winner for a given input, we designed an activation function such that its outcome is greatest for the closest "responsible" unit. Responsible units 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

**Input:** $\mathbf{x}, \mathbf{y} \in \mathfrak{C}_0^m$
**Output:** Activation $d_{\mathfrak{C}}(\mathbf{x}, \mathbf{y}) \in (\mathbb{N}, \{1, 2, 3\}, \mathbb{R})$

**if** $\mathbf{x} = \mathbf{y}$ **then return** $(\infty, 0, 0)$
$l :=$ level of $H_{in}(\{\mathbf{x}, \mathbf{y}\})$
Compute $k$ according to the following 3 cases:

$$k := \begin{cases} 3 & \text{if } H_{in}(\{\mathbf{x}\}) \text{ and } H_{in}(\{\mathbf{y}\}) \text{ are of level greater than } l \\ 2 & \text{if } H_{in}(\{\mathbf{x}\}) \text{ or } H_{in}(\{\mathbf{y}\}) \text{ is of level greater than } l \\ 1 & \text{otherwise} \end{cases}$$

$m := \frac{1}{|\mathbf{x}-\mathbf{y}|}$, i.e., $m$ is the inverse of the Euclidean distance
**return** $(l, k, m)$

**Fig. 9.11.** Algorithm for the activation function for the Fine Blend

*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 function $d_{\mathfrak{C}}$ are given in Algorithm 9.11. 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 activations this first component is equal, the second component is used and the third, if the first two are equal.

*Example 11.* Let $v_1, v_2, v_3$ and $v_4$ from Example 10 be the incoming connections for the units $u_1, u_2, u_3$ and $u_4$ respectively. The different input regions for which each of the units are responsible are depicted in Figure 9.12. For the vector $i = (0.05, 0.02)$ (also shown in Figure 9.12), we obtain the following activations:

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

I.e., 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. This is due to the fact, that 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.

### 9.4.4 Training

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

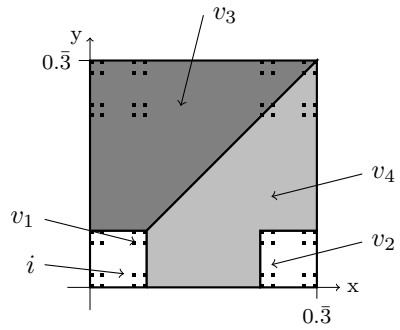


**Fig. 9.12.** The areas of responsibility for $v_1$, $v_2$, $v_3$ and $v_4$. For each of the four regions one of the units is responsible.

training samples with input and desired output. This process can be used to refine a network resulting from an incorrect program, or to train a network from scratch.[7] 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 [37]. For a more detailed discussion of the training algorithms and modifications we refer to [35, 36].

### Adapting the weights

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

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

Furthermore, we change the output weights for $u$ towards the desired output:

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

$\eta$ and $\mu$ are predefined learning rates. Note that (in contrast to the methods described in [37]) 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 9.13 depicts the adaptation of the incoming connections.

[7] E.g., using an initial network with a single hidden layer unit.

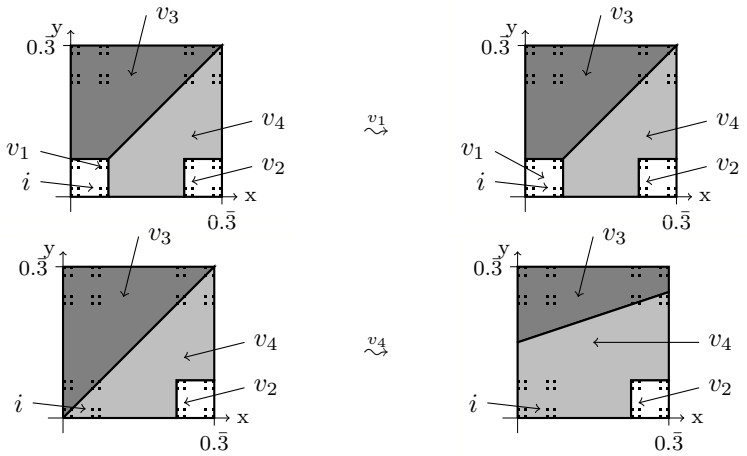


**Fig. 9.13.** The adaptation of the input weights for a given input $i$. The first row shows the result of adapting $v_1$. The second row shows the result if $v_1$ would not be there and therefore $v_4$ would be selected as winner. To emphasize the effect, we used a learning rate $\mu = 1.0$, i.e., 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 it is winner for. 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 9.14.

### 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

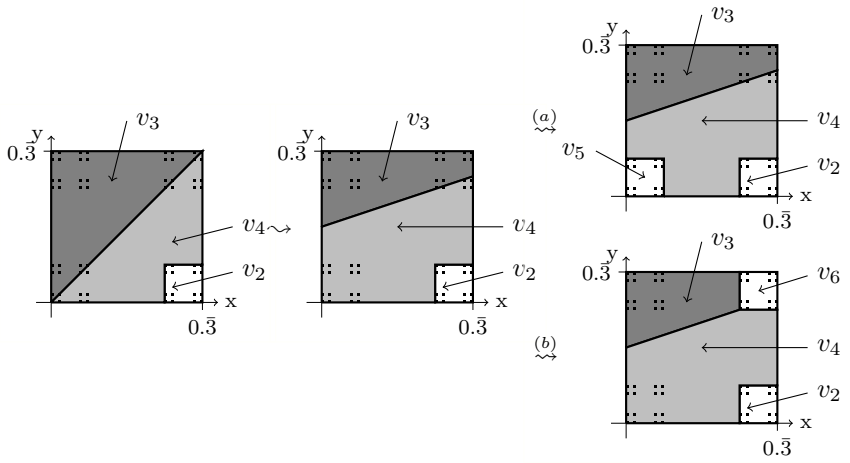


**Fig. 9.14.** Adding a new unit to support $v_4$. First, $v_4$ is moved to the center of the hyper-square it is responsible for. There are four possible sub-hyper-squares to add a new unit. Because $v_4$ is neither responsible for the north-western, nor for the south-eastern sub-hyper-square, there 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.

contribution of a unit is the expected increase of error if the unit would be removed [37]. If a unit's utility drops below a threshold, the unit will be removed as depicted in Figure 9.15.

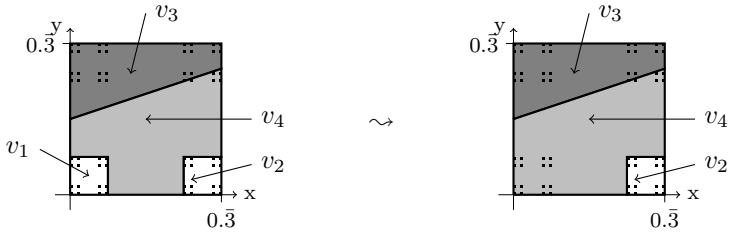


**Fig. 9.15.** Removing an inutile unit. Let us assume that the outgoing weights of $v_1$ and $v_4$ are equal. In this case we would find that the over-all error would not increase if we remove $v_1$. Therefore its utility would decrease over time until it drops below the threshold and the unit is removed.

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 [37], 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 in the following section, will show that our method actually works.

## 9.5 Evaluation

In this section we will discuss some preliminary experiments. Those are not meant to be exhaustive, but rather to provide a proof of concept. An in-depth anaylsis of all required parameters will be done in the future. In the diagrams, we use a logarithmic scale for the error axis, and the error values are relative to $\varepsilon$, i.e. a value of 1 designates an absolute error of $\varepsilon$.

To initialize the network we used the following wrong program:

$$e(s(X)) \leftarrow \neg o(X).$$
$$o(X) \leftarrow e(X).$$

Training samples were created randomly using the semantic operator of the correct program given in Example 2, viz.:

$$e(0).$$
$$e(s(X)) \leftarrow o(X).$$
$$o(X) \leftarrow \neg e(X).$$

### Variants of Fine Blend

To illustrate the effects of varying the parameters, we use two setups: One with softer utility criteria (FineBlend 1) and one with stricter ones (FineBlend 2). Figure 9.16 shows that, 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. 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 will be investigated in the future. For our further experiments we will use the FineBlend 1 parameters, which resulted from a mixture of intuition and (non-exhaustive) comparative simulations.

### Fine Blend versus SGNG

Figure 9.17 compares FineBlend 1 with SGNG [37]. Both start off similarly, but soon SGNG fails to improve further. The increasing number of units is partly due to the fact that no error threshold is used to inhibit refinement, but this should not be the cause for the constantly high error level. The choice of

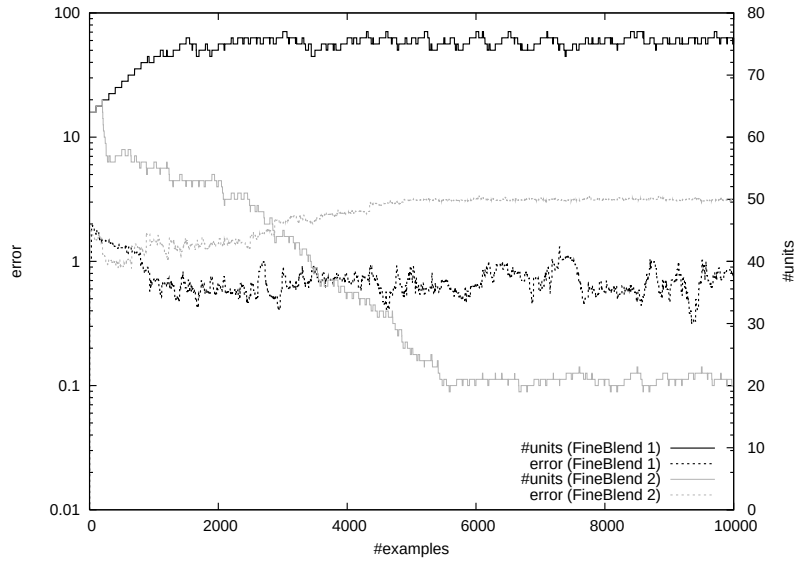


**Fig. 9.16.** FineBlend 1 versus FineBlend 2.

SGNG parameters is rather subjective, and even though some testing was done to find them, they might be far from optimal. Finding the optimal parameters for SGNG is beyond the scope of this paper; however, it should be clear that it is not perfectly suited for our specific application. This comparison to an established generic architecture shows that our specialized architecture actually works, i.e. it is able to learn, and that it achieves the goal of specialization, i.e. it outperforms the generic architecture in our specific setting.

### 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.

### Unit Failure

Figure 9.18 shows the effects of unit failure. A FineBlend 1 network is initialized and refined through training with 5000 samples, then one third of its

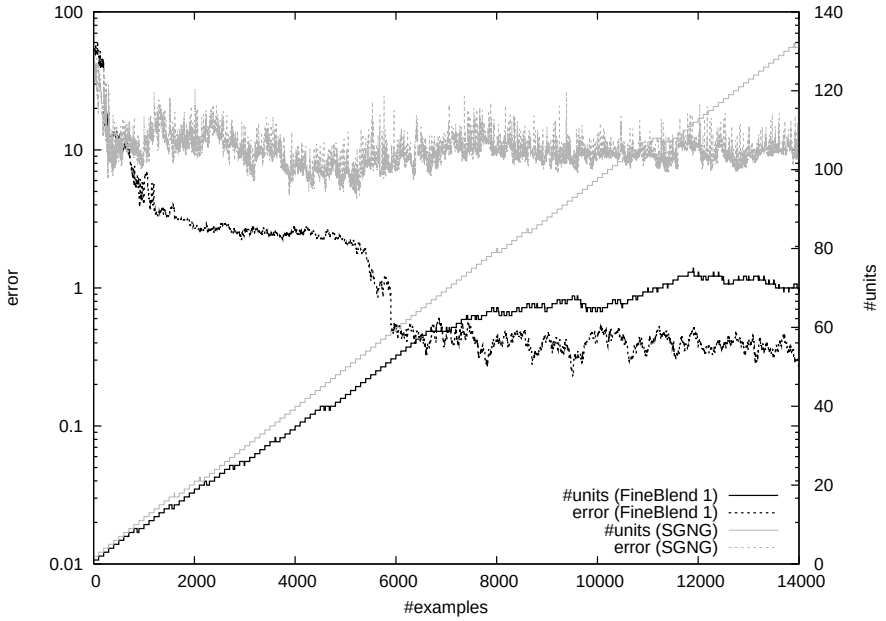


**Fig. 9.17.** FineBlend 1 versus SGNG.

hidden units are removed randomly, and then training is continued as if nothing had happened. The network proves to handle the damage gracefully and to recover quickly. The relative error exceeds 1 only slightly and drops back very soon; the number of units continues to increase to the previous level, recreating the redundancy necessary for robustness.

### Iterating Random Inputs

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 desired model of the program, or more precisely to a fixed point of the program's single-step operator. In this sense, the Core Method allows to reason with the acquired knowledge. For our system, this model generation also serves as a sanity check: if the model can be reproduced successfully in the connectionist setting then this shows that the network was indeed able to acquire symbolic knowledge during training.

In our running example program, a unique fixed point is known to exist. To check whether our system reflects this, we proceed as follows:

1. Train a network from scratch until the relative error caused by the network is below 1, i.e. network outputs are in the $\varepsilon$-neighborhood of the desired outputs.

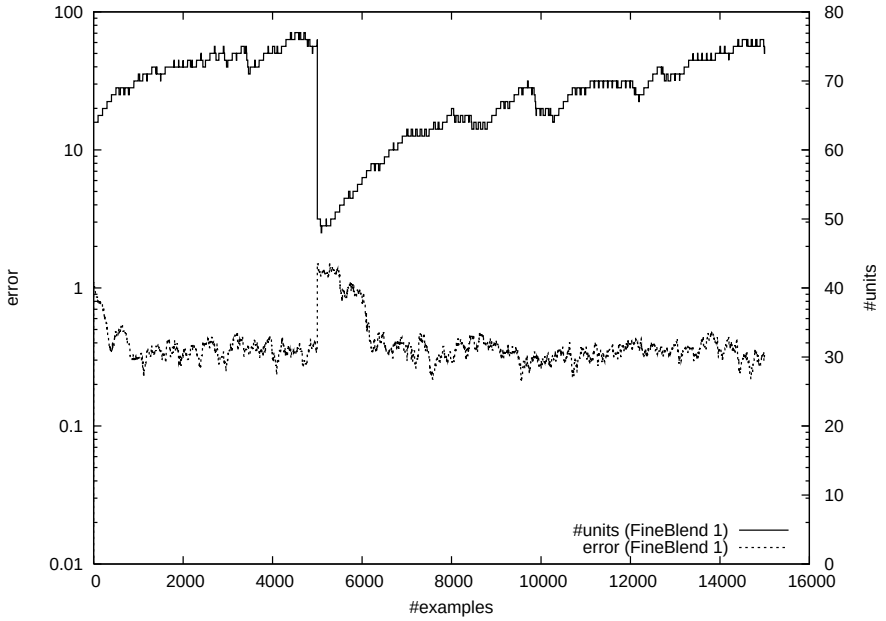


**Fig. 9.18.** The effects of unit failure.

2. Transform the obtained network into a recurrent one by connecting the outputs to the corresponding inputs.
3. 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.
4. Iterate the network until it reaches a stable state, i.e. until the outputs stay inside an $\varepsilon$-neighborhood.

For our example program, the unique fixed point of $T_{\mathcal{P}_2}$ is $M_2$ as given in Example 4. Figure 9.19 shows the input space and the $\varepsilon$-neighborhood of $M$, along with all intermediate results of the iteration for 5 random initial inputs. The example computations converge, because the underlying program is acyclic [35, 36, 29]. After at most 6 steps, the network is stable in all cases, in fact it is completely stable in the sense that all outputs stay exactly the same and not only within an $\varepsilon$-neighborhood. This corresponds roughly to the number of applications of our program's $T_{\mathcal{P}_2}$ 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.

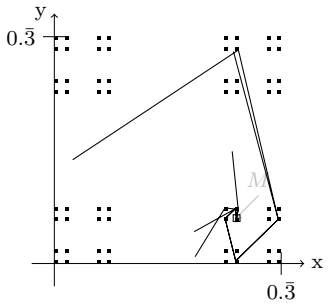


**Fig. 9.19.** Iterating random inputs. The two dimensions of the input vectors are plotted against each other. The $\varepsilon$-neighborhood of the fixed point $M$ is shown as a small box.

## 9.6 Conclusion

After reviewing the connection between the $T_{\mathcal{P}}$-operator associated with propositional logic programs and simple three-layer feed-forward neural networks, we reported on extensions to first order logic programs. By restating results from [29], we showed that a representation of semantic operators is possible. Afterwards, we described a first approach [32] to actually construct approximating networks. This approach was extended to a multi-dimensional setting allowing for arbitrary precision, even if implemented on a real computer [35, 36]. Finally, we reported on some prelimenary experiments which show that our approach actually works.

Our system realises part of the neural-symbolic cycle in that it is able to learn first-order logic programs, and to outperform other approaches in this specific learning task. The trained network is also able to generate the model of the target program which shows that it has acquired the desired declarative knowledge. At the same time, the network is robust in that it can recover from substantial unit failure. Summarising, our system combines the power of connectionist learning and robustness with the processing of declarative knowledge, and thus retains the best of both worlds. It is fair to say that this system provides a major step in the overcoming what McCarthy called *propositional fixation*.

We are currently re-implementing the first-order Core Method in order to further evaluate and test it using real world examples. An in-depth analysis of the system shall provide heuristics to determine optimal values for the parameters of the system. Furthermore, we intend to compare our approach with other methods. Concerning a complete neural-symbolic cycle we note that whereas the extraction of propositional rules from trained networks is well understood, the extraction of first-order rules is still an open question, which will be addressed in the future.

**Acknowledgments:**

Many thanks to Sven-Erik Bornscheuer, Artur d'Avila Garcez, Yvonne McIntyre (formerly Kalinke), Anthony K. Seda, Hans-Peter Störr and Jörg Wunderlich who all contributed to the Core Method.

## References

1. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics **5** (1943) 115–133
2. Pinkas, G.: Symmetric neural networks and logic satisfiability. Neural Computation **3** (1991) 282–291
3. McCarthy, J.: Epistemological challanges for connectionism. Behavioural and Brain Sciences **11** (1988) 44 Commentary to [12].
4. Bader, S., Hitzler, P.: Dimensions of neural-symbolic integration - a structured survey. In: S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb and J. Woods (eds).: We Will Show Them: Essays in Honour of Dov Gabbay, Volume 1. International Federation for Computational Logic, College Publications (2005) 167–194
5. Ballard, D.H.: Parallel logic inference and energy minimization. In: Proceedings of the AAAI National Conference on Artificial Intelligence. (1986) 203 – 208
6. Lange, T.E., Dyer, M.G.: High-level inferencing in a connectionist network. Connection Science **1** (1989) 181–217
7. Shastri, L., Ajjanagadde, V.: From associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. Behavioural and Brain Sciences **16** (1993) 417–494
8. Hölldobler, S., Kurfess, F.: CHCL – A connectionist inference system. In Fronhöfer, B., Wrightson, G., eds.: Parallelization in Inference Systems. Springer, LNAI *590* (1992) 318 – 342
9. Pollack, J.B.: Recursive distributed representations. AIJ **46** (1990) 77–105
10. Plate, T.A.: Holographic reduced networks. In Giles, C.L., Hanson, S.J., Cowan, J.D., eds.: Advances in Neural Information Processing Systems 5. Morgan Kaufmann (1992)
11. Elman, J.L.: Structured representations and connectionist models. In: Proceedings of the Annual Conference of the Cognitive Science Society. (1989) 17–25
12. Smolensky, P.: 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)
13. Bader, S., Hitzler, P., Hölldobler, S.: The Integration of Connectionism and First-Order Knowledge Representation and Reasoning as a Challenge for Artificial Intelligence. Information **9** (2006).
14. Towell, G.G., Shavlik, J.W.: Extracting refined rules from knowledge-based neural networks. Machine Learning **13** (1993) 71–101
15. Hölldobler, S., Kalinke, Y.: Towards a massively parallel computational model for logic programming. In: Proceedings ECAI94 Workshop on Combining Symbolic and Connectionist Processing, ECCAI (1994) 68–77

16. Bishop, C.M.: Neural Networks for Pattern Recognition. Oxford University Press (1995)
17. Lloyd, J.W.: Foundations of Logic Programming. Springer, Berlin (1988)
18. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, Los Altos, CA (1988) 89–148
19. Hitzler, P., Hölldobler, S., Seda, A.K.: Logic programs and connectionist networks. Journal of Applied Logic **3** (2004) 245–272
20. d'Avila Garcez, A.S., Zaverucha, G., de Carvalho, L.A.V.: Logical inference and inductive learning in artificial neural networks. In Hermann, C., Reine, F., Strohmaier, A., eds.: Knowledge Representation in Neural networks. Logos Verlag, Berlin (1997) 33–46
21. d'Avila Garcez, A.S., Broda, K.B., Gabbay, D.M.: Neural-Symbolic Learning Systems — Foundations and Applications. Perspectives in Neural Computing. Springer, Berlin (2002)
22. Kalinke, Y.: Ein massiv paralleles Berechnungsmodell für normale logische Programme. Master's thesis, TU Dresden, Fakultät Informatik (1994) (in German).
23. Seda, A., Lane, M.: Some aspects of the integration of connectionist and logic-based systems. In: Proceedings of the Third International Conference on Information, International Information Institute, Tokyo, Japan (2004) 297–300
24. d'Avila Garcez, A.S., Lamb, L.C., Gabbay, D.M.: A connectionist inductive learning system for modal logic programming. In: Proceedings of the IEEE International Conference on Neural Information Processing ICONIP'02, Singapore. (2002)
25. d'Avila Garcez, A.S., Lamb, L.C., Gabbay, D.M.: Neural-symbolic intuitionistic reasoning. In A. Abraham, M.K., Franke, K., eds.: Frontiers in Artificial Intelligence and Applications, Melbourne, Australia, IOS Press (2003) Proceedings of the Third International Conference on Hybrid Intelligent Systems (HIS'03).
26. Andrews, R., Diederich, J., Tickle, A.: A survey and critique of techniques for extracting rules from trained artificial neural networks. Knowledge–Based Systems **8** (1995)
27. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Networks **2** (1989) 359–366
28. Funahashi, K.I.: On the approximate realization of continuous mappings by neural networks. Neural Networks **2** (1989) 183–192
29. Hölldobler, S., Kalinke, Y., Störr, H.P.: Approximating the semantics of logic programs by recurrent neural networks. Applied Intelligence **11** (1999) 45–58
30. Willard, S.: General Topology. Addison–Wesley (1970)
31. Fitting, M.: Metric methods, three examples and a theorem. Journal of Logic Programming **21** (1994) 113–127
32. Bader, S., Hitzler, P., Witzel, A.: Integrating first-order logic programs and connectionist systems – a constructive approach. In d'Avila Garcez, A.S., Elman, J., Hitzler, P., eds.: Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK. (2005)
33. Witzel, A.: Integrating first-order logic programs and connectionist systems – a constructive approach. Project thesis, Department of Computer Science, Technische Universität Dresden, Dresden, Germany (2005)
34. Barnsley, M.: Fractals Everywhere. Academic Press, San Diego, CA, USA (1993)

35. Witzel, A.: Neural-symbolic integration – constructive approaches. Master's thesis, Department of Computer Science, Technische Universität Dresden, Dresden, Germany (2006)
36. Bader, S., Hitzler, P., Hölldobler, S., Witzel, A.: A fully connectionist model generator for covered first-order logic programs. In Veloso, M.M., ed.: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India, Menlo Park CA, AAAI Press (2007) 666–671
37. Fritzke, B.: Vektorbasierte Neuronale Netze. Habilitation, Technische Universität Dresden (1998)