Neural-Symbolic Integration

Constructive Approaches

Andreas Witzel, Technische Universität Dresden

Abstract

The field of neural-symbolic integration has received much attention recently. While with propositional paradigms, the integration of symbolic knowledge and connectionist systems (also called artificial neural networks) has already resulted in applicable systems, the theoretical foundations for the first-order case are currently being laid and first perspectives for real implementations are emerging. Two important components of the neural-symbolic learning cycle [BH05] are representation, i.e. encoding symbolic knowledge into connectionist systems, and training, i.e. adjusting these connectionist systems according to information observed in other ways. These components are the focus of this thesis. Extending results from [Wit05, BHW05], a practically feasible way is presented to approximate and embed the semantic operator of covered logic programs in a real-valued domain, and connectionist architectures suitable for representing this particular form of symbolic knowledge are developed and evaluated along with appropriate training methods.

Contents

1 Introduction

Logic programs have been studied thoroughly in computer science and artificial intelligence and are well understood. They are human-readable, they basically consist of logic formulae, and there are well-founded mathematical theories defining exactly the meaning of a logic program. Logic programs thus constitute one of the most prominent paradigms for knowledge representation and reasoning. But there is also a major drawback: Logic programming is unsuitable for certain learning tasks, in particular in the full first-order case.

On the other hand, for connectionist systems — also called artificial neural networks — there are established and rather simple training or learning algorithms. But it is hard to manually construct a connectionist system with a desired behaviour, and even harder to find a declarative interpretation of what a given connectionist system does. Connectionist systems perform very well in certain settings, but in general we do not understand why or how.

Thus, logic programs and connectionist systems have contrasting advantages and disadvantages. It would be desirable to integrate both approaches in order to combine their respective advantages while avoiding the disadvantages. We could then train a connectionist system to fulfil a certain task, and afterwards translate it into a logic program in order to understand it or to prove that it meets a given specification. Or we might write a logic program and turn it into a connectionist system which could then be optimised using a training algorithm. See [BH05] for a recent survey putting these tasks into a broader context.

Main challenges for the integration of symbolic and connectionist knowledge thus centre around the questions (1) how to extract logical knowledge from trained connectionist systems, and (2) how to encode symbolic knowledge within such systems. We find it natural to start with (2), hoping that extraction methods can be deduced from successful methods for encoding.

For propositional logic programs, encodings into connectionist systems like [HK94] led immediately to applicable algorithms. Corresponding learning paradigms have been developed [GZ99, GBG02] and applied to real settings.

For the first-order logic case, however, the situation is much more difficult, as laid out in [BHH04]. Concrete translations, as in [BH04, BGH05], yield non-standard[1] network architectures. For standard architectures, previous work has established non-constructive proofs showing the existence of connectionist systems which approximate given logic programs with arbitrary precision [HKS99, HHS04].

[1]standard in the sense of [BH05]

In [Wit05, BHW05] we have presented a construction for such connectionist systems; however, with the conclusion that it is not suitable for a real implementation. In this paper, we will generalise these results, present and discuss various standard network architectures and appropriate training methods, and evaluate those selected for implementation.

First, in Section 2, we will give a short introduction to logic programs and connectionist systems. In Section 3, we will then generalise the basic definitions and results obtained in [Wit05, BHW05]. Section 4 will present two attempts to construct feed-forward networks with sigmoidal output functions in the hidden layers, coming to the conclusion that other types of networks are better suited for our purposes. In Section 5, we introduce the general network structure and the training framework common to all architectures we implemented. Section 6 then defines these architectures, specifies concrete transformation and training methods, and discusses their properties. In Section 7, we evaluate the resulting systems using test scenarios, and Section 8 concludes the paper.

2 Preliminaries

In this section, we shortly review the basic notions needed from logic pogramming and connectionist systems. Main references for background reading are [Llo88] and [Roj96], respectively.

2.1 Logic Programs

A *logic program* over some first-order language $\mathcal{L}$ is a set of (implicitly universally quantified) *clauses* of the form $A \leftarrow L_1 \wedge \cdots \wedge L_n$, where $n \in \mathbb{N}$ may differ for each clause, A is an *atom* in $\mathcal{L}$ with variables from a set $\mathcal{V}$, 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. As an abbreviation, we will sometimes replace $L_1 \wedge \cdots \wedge L_n$ by body and write $A \leftarrow \mathrm{body}$. 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.

Example 2.1. *The following is a covered logic program which will serve as our running example. The intended meaning of the clauses is given to the right.*

$e(0).$	*% 0 is even*
$o(X) \leftarrow \neg e(X).$	*% X is odd if it is not even*
$e(s(X)) \leftarrow o(X).$	*% the successor $s(X)$ of an odd X is even*

The *Herbrand universe* $\mathcal{U}_P$ is the set of all ground terms of $\mathcal{L}$, the *Herbrand base* $\mathcal{B}_P$ is the set of all ground atoms, which we assume to be infinite. Indeed the case when $\mathcal{B}_P$ is finite is of limited interest to us as it reduces to a propositional setting as studied in [HK94, GZ99]. A *ground instance* of a literal or a clause is obtained by replacing all variables by terms from $\mathcal{U}_P$. For a logic program P, $\mathcal{G}(P)$ is the set of all ground instances of clauses from P.

A *level mapping* is a function $\|\cdot\| : \mathcal{B}_P \to \mathbb{N} \setminus \{0\}$. The *level* of an atom A is denoted by $\|A\|$. The level of a literal is that of the corresponding atom.

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

Example 2.2. *For the program from Example 2.1, we have:*

$$\begin{aligned}
\mathcal{U}_P &= \{0, s(0), s(s(0)), \dots\} \\
\mathcal{B}_P &= \{e(0), o(0), e(s(0)), o(s(0)), \dots\} \\
\mathcal{G}(P) &= \left\{ \begin{array}{r@{}l} e(0). & \\ o(0) &{}\leftarrow \neg e(0). \\ e(s(0)) &{}\leftarrow o(0). \\ o(s(0)) &{}\leftarrow \neg e(s(0)). \\ e(s(s(0))) &{}\leftarrow o(s(0)). \\ & \vdots \end{array} \right\}
\end{aligned}$$

We will use the usual abbreviation $s^n(0)$ for the n-fold application of s to 0. With $\|e(s^n(0))\| := 2n + 1$ and $\|o(s^n(0))\| := 2n + 2$, we find that P is acyclic.

A *(Herbrand) interpretation* is a subset I of $\mathcal{B}_P$. Those atoms A with $A \in I$ are said to be *true*, or to *hold*, under I (in symbols: $I \models A$), those with $A \notin I$ are said to be *false*, or to *not hold*, under I (in symbols: $I \not\models A$). $\mathcal{I}_P = 2^{\mathcal{B}_P}$ is 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 $A \leftarrow \mathrm{body} \in \mathcal{G}(P)$ in the usual sense. That is, if of all body literals I contains exactly those which are not negated (i.e. $I \models \mathrm{body}$), then I must also contain the head.

Example 2.3. *Consider these three Herbrand interpretations for P from Example 2.1:*

$$\begin{aligned}
I_1 &= \{e(0), o(s(0)), e(s^2(0)), o(s^3(0))\} \\
I_2 &= \{e(0), e(s^2(0)), o(s^3(0)), e(s^3(0))\} \\
I_3 &= \mathcal{B}_P
\end{aligned}$$

$I_1 \not\models P$ since $e(s^4(0)) \leftarrow o(s^3(0)) \in \mathcal{G}(P)$ and $o(s^3(0)) \in I_1$, but $e(s^4(0)) \notin I_1$. $I_2 \not\models P$ since $o(s(0)) \leftarrow \neg e(s(0)) \in \mathcal{G}(P)$ and $e(s(0)) \notin I_2$, but $o(s(0)) \notin I_2$. $I_3 \models P$.

The *single-step operator* $T_P : \mathcal{I}_P \to \mathcal{I}_P$ maps an interpretation I to the set of exactly those atoms A for which there is a clause $A \leftarrow \mathrm{body} \in \mathcal{G}(P)$ with $I \models \mathrm{body}$. 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 [ABW88]. The well-known stable models [GL88], for example, are always supported. In Example 2.1, the (obviously intended) model $\{e(0), o(s(0)), e(s^2(0)), o(s^3(0)), e(s^4(0)), \dots\}$ is supported (and stable), while I_3 is a model but not supported.

Example 2.4. *For P from Example 2.1 and I_2 from Example 2.3, we get the following by successive application (i.e. iteration) of T_P:*

$$\begin{aligned}
I_2 &\overset{T_P}{\mapsto} \{e(0), o(s(0)), e(s^4(0)), o(s^4(0)), o(s^5(0)), o(s^6(0)), \dots\}\\
&\overset{T_P}{\mapsto} \{e(0), o(s(0)), e(s^2(0)), o(s^2(0)), o(s^3(0)), e(s^5(0)), o(s^5(0)) \dots\}\\
&\overset{T_P}{\mapsto} \{e(0), o(s(0)), e(s^2(0)), e(s^3(0)), o(s^3(0)), e(s^4(0)), o(s^4(0)), e(s^6(0)), \dots\}\\
&\overset{T_P}{\mapsto} \dots
\end{aligned}$$

For a certain class of programs, the process of iterating T_P can be shown to converge[2] to the unique supported Herbrand model of the program, which in this case is the model describing the semantics of the program [HS03]. This class is described by the fact that T_P is a contraction with respect to a certain metric. A more intuitive description remains to be found, but at least all acyclic programs[3] are contained in this class. That is, given some acyclic program P, we can find its unique supported Herbrand model by iterating T_P and computing a limit. In Example 2.4 for instance, the iteration converges in this sense to $\{e(0), o(s(0)), e(s^2(0)), o(s^3(0)), e(s^4(0)), \dots\}$, which is the unique supported model of the program.

2.2 Connectionist Systems

A *connectionist system* — or *artificial neural network* — is a complex network of simple computational units, also called *nodes* or *neurons*, which accumulate real numbers from their inputs and send a real number to their output. Each unit's output is *connected* to other units' inputs with a certain real-numbered *weight*. Those units without incoming connections are called *input units* or *input neurons*, those without outgoing ones are called *output units* or *output neurons*.

We will exclusively deal with layered feed-forward networks, i.e. networks without cycles where the outputs of units in one layer are only connected to the inputs of units in the next layer. The first and last layers contain the input and output units respectively, the remaining layers are called *hidden layers*. In the input layer, besides actual input units feeding input from the outside world into the network, we allow additional units which constantly output the value 1.

Each unit has an *input function* which uses the connections' weights to merge its inputs into one single value called *activation* or *potential*, and an *output function* which

[2]Convergence in this case is convergence with respect to the Cantor topology on $\mathcal{I}_P$, or equivalently, with respect to a natural underlying metric. For further details, see [HS03], where also a general class of programs, called Φ-*accessible programs*, is described, for which iterating T_P always converges in this sense.

[3]in this case the level mapping does not need to be injective

then computes the output. If a unit has inputs $x_1, \ldots, x_n$ with weights $w_1, \ldots, w_n$, then the *weighted sum* input function is $\sum_{i=1}^{n} x_i w_i$. An example for a locally receptive *radial basis* input function is $\sqrt{\sum_{i=1}^{n}(x_i - w_i)^2}$, but any function which can be interpreted as computing some kind of distance is suitable here.

Example 2.5. *The connectionist system shown in Figure 1 classifies its input by outputting* 0 *for inputs* ≤ 5, 0.5 *for inputs* > 5 *and* ≤ 7, *and* 1 *for inputs* > 7.

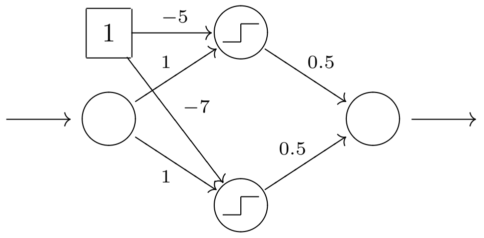


Figure 1: The classifying connectionist system from Example 2.5, depicted as a graph of units with connections. The *weights* of the connections are denoted at the arrows representing them. The *input* resp. *output units* are marked by an unconnected incoming resp. outgoing arrow. The constant 1 unit is denoted as a square. In the hidden layer, we use *weighted sum input functions* along with *step output functions* which output 0 for activations ≤ 0 and 1 for activations > 0. The output unit simply sums up its weighted inputs. It can be regarded as a unit with weighted sum input function and identity output function.

Given a connectionist system, it is hard to read off any meaning beyond the obvious purely mathematical meaning, for example some kind of symbolic or logical interpretation. Vice versa, given some description of desired behaviour, it is not straightforward to design a corresponding connectionist system.

One of the main advantages of connectionist systems is that there are established learning algorithms which can be used to train or adapt existing systems and perform remarkably well. A prominent method is the *backpropagation algorithm*. It changes the network's parameters and performs a gradient descent in order to minimise the deviation from a given desired output. Thus, in order for this learning method to be applicable, the function computed by the network needs to be differentiable. Obviously, if all input and output functions are differentiable, then so is the whole network function since it can be seen as a function composition.

But even if standard learning algorithms cannot be applied, devising training methods for connectionist systems can be much simpler than for symbolic systems. In this thesis, we will try both approaches.

3 Multi-Dimensional Approximations of Logic Programs

In [Wit05, BHW05] we reviewed a method introduced in [HKS99] to embed Herbrand interpretations into $\mathbb{R}$ and to represent the single-step operator of covered logic programs as real-valued function on $\mathbb{R}$. We also developed approximations of this function with arbitrary precision. These approximations consisted of finitely many constant pieces, which was the basis for the transformation into connectionist systems. One of our conclusions was that, in order to cope with the limited precision in an implementation, it would be necessary to distribute the representations of interpretations on multiple real numbers. This multi-dimensional extension is the subject of this section.

In Section 3.1, we will fix some notation and straightforwardly generalise the necessary basic definitions from Section 2.1 and results from [HKS99, HHS04]. In the slightly more involved Section 3.2, we will then extend the actual approximation and prove some properties which we will use later on.

3.1 Basic Definitions

Definition 3.1 (Notation). Throughout this paper, we will use m to denote the number of dimensions on which interpretations are distributed, and j to refer to a specific dimension. We will denote m-dimensional vectors in bold face and use normal face with indices $1, \ldots, m$ to access the components. In particular, bold face constants denote m-dimensional vectors of the respective constant, e.g. $\mathbf{0} = (0, \ldots, 0)$. Vector comparisons and similar operations as well as functions on $\mathbb{R}$ are always component-wise, e.g.

$$\begin{aligned} \boldsymbol{x} \leq \boldsymbol{y} \quad &\text{iff} \quad \bigwedge_{j=1}^{m} x_j \leq y_j \\ \max\{\boldsymbol{x}, \boldsymbol{y}\} &:= (\max\{x_1, y_1\}, \ldots, \max\{x_m, y_m\}) \\ \max \boldsymbol{x} &:= \max_{1 \leq j \leq m} \{x_j\} \\ f(\boldsymbol{x}) &:= (f(x_1), \ldots, f(x_m)) \quad \text{for } f : \mathbb{R} \to \mathbb{R} \\ \text{e.g. } |\boldsymbol{x}| &:= (|x_1|, \ldots, |x_m|). \end{aligned}$$

Note that $\max$ is not treated as a function from $\mathbb{R}$ to $\mathbb{R}$, which intuitively would not make sense anyway.
Additionally, we define component-wise multiplication, division, and exponentiation:

$$\begin{aligned} \boldsymbol{x} * \boldsymbol{y} &:= (x_1 \cdot y_1, \ldots, x_m \cdot y_m) \\ \frac{\boldsymbol{x}}{\boldsymbol{y}} := \boldsymbol{x}/\boldsymbol{y} &:= (x_1/y_1, \ldots, x_m/y_m) \\ z^{\boldsymbol{x}} &:= (z^{x_1}, \ldots, z^{x_m}) \end{aligned}$$

Floating point numbers to a given base b are denoted by an appended subscript b.

Definition 3.2 (Multi-Dimensional Level Mappings). An *m-dimensional level mapping* is a tuple of functions $(\|\cdot\|, \dim)$ where $\|\cdot\|$ is a level mapping and $\dim$ yields a dimension for each atom, i.e.

$$\|\cdot\| : \mathcal{B}_P \to \mathbb{N} \setminus \{0\}$$

and

$$\dim : \mathcal{B}_P \to \{1, \dots, m\}.$$

For an atom $A \in \mathcal{B}_P$, $\|A\|$ and $\dim(A)$ are the *level* and the *dimension* of A. The level and the dimension of a literal are those of the corresponding atom.

$(\|\cdot\|, \dim)$ is said to be *bijective* iff for each dimension $j \in \{1, \dots, m\}$, $\|\cdot\|$ restricted to $\dim^{-1}(j)$ is bijective. In this paper, we will always assume $(\|\cdot\|, \dim)$ to be bijective.

Definition 3.3 (Acyclicity wrt multi-dimensional level mappings). A logic program P is *acyclic* with respect to a multi-dimensional level mapping $(\|\cdot\|, \dim)$ if for all clauses $A \leftarrow L_1 \wedge \cdots \wedge L_n \in \mathcal{G}(P)$ and all $1 \leq i \leq n$, we have that

$$\begin{aligned} &\|A\| > \|L_i\| \text{ or} \\ &\|A\| = \|L_i\| \text{ and } \dim(A) > \dim(L_i) \end{aligned}$$

Example 3.4. *With*

$$\begin{aligned} \|e(s^n(0))\| &:= n+1 & \qquad \dim(e(s^n(0))) &:= 1 \\ \|o(s^n(0))\| &:= n+1 & \qquad \dim(o(s^n(0))) &:= 2 \end{aligned}$$

for all $n \geq 0$, we find that our running Example 2.1 is acyclic. We will always use this level mapping for the running example.

Lemma 3.5. A logic program P is acyclic wrt to some level mapping iff it is acyclic wrt to some multi-dimensional level mapping.

Proof.

"$\Rightarrow$": Assume P is acyclic wrt $\|\cdot\|$. With $\dim(A) := 1$ for all $A \in \mathcal{B}_P$, P is acyclic wrt $(\|\cdot\|, \dim)$.

"$\Leftarrow$": Assume P is acyclic wrt the m-dimensional level mapping $(\|\cdot\|, \dim)$, then P is acyclic wrt to the level mapping

$$\|\cdot\|' : A \mapsto m \cdot (\|A\| - 1) + \dim(A) \quad \text{for all } A \in \mathcal{B}_P,$$

since for any atom A and literal L

(i)

$$\begin{aligned} &\|A\| > \|L\| \\ \Rightarrow\ & m \cdot (\|A\| - 1) \geq m \cdot (\|L\| - 1) + m && (\|A\|, \|L\| \in \mathbb{N}) \\ \Rightarrow\ & \|A\|' > \|L\|' && (\dim(A) \geq 1, \dim(L) \leq m) \end{aligned}$$

(ii)

$$\|A\| = \|L\| \wedge \dim(A) > \dim(L) \Rightarrow \|A\|' > \|L\|'$$

□

Note that bijectivity is preserved in both directions in this proof. Furthermore, under our standing condition that multi-dimensional level mappings are bijective, all acyclic programs with respect to some multi-dimensional level mapping are covered.

In the following definitions, we extend the standard technique for bridging the symbolic world of logic programs with the real-numbers-based world of connectionist systems, namely the embedding of the single-step operator, which carries the meaning of a logic program, into the real numbers as established for this purpose in [HKS99].

Definition 3.6 (Multi-Dimensional Embedding of $\mathcal{B}_P$). The *m-dimensional embedding $\boldsymbol{R} : \mathcal{B}_P \to \mathbb{R}^m$*, is defined as

$$\boldsymbol{R}(A) := (R_1(A), \dots, R_m(A)) \qquad \text{for } A \in \mathcal{B}_P$$

where

$$R_j(A) := \begin{cases} b^{-\|A\|} & \text{if } j = \dim(A) \\ 0 & \text{otherwise.} \end{cases} \qquad \text{with some (fixed) base } b \geq 3$$

It is extended to $\mathcal{I}_P$ by setting

$$R_j(I) := \sum_{A \in I} R_j(A) \qquad \text{for } I \in \mathcal{I}_P$$

Since we only deal with bijective level mappings, the R_j are injective, and thus so is $\boldsymbol{R}$. In particular, for $I \in \mathcal{I}_P$ with $\boldsymbol{R}(I) = \boldsymbol{x}$ we have that

$$I = \boldsymbol{R}^{-1}(\boldsymbol{x}) = \bigcup_{j=1}^{m} R_j^{-1}(x_j).$$

The higher the level of an atom, the lower its embedded value is. This property gives more weight to atoms of lower level. In our running example, this can be interpreted such that atoms of lower nesting depth are considered to be more important.

Example 3.7. *With the level mapping defined in Example 3.4, we obtain the following values for the embedding of the interpretations from Example 2.3:*

$$\begin{aligned} \boldsymbol{R}(I_1) &= (0.1010_b, 0.0101_b) \\ \boldsymbol{R}(I_2) &= (0.1011_b, 0.0001_b) \\ \boldsymbol{R}(I_3) &= (0.1111\ldots_b, 0.1111\ldots_b) \end{aligned}$$

Note: With base 2 we would lose injectivity, since for instance the number $0.01111\ldots_2$ has the same value as 0.1_2.

Definition 3.8 (Multi-Dimensional Embedding of T_P). The *m-dimensional embedding of T_P, $\boldsymbol{f}_P : D_{\boldsymbol{f}} \to D_{\boldsymbol{f}}$* with $D_{\boldsymbol{f}} := \{\boldsymbol{R}(I) | I \in \mathcal{I}_P\} \subseteq \mathbb{R}^m$, is defined as

$$\boldsymbol{f}_P(\boldsymbol{x}) := \boldsymbol{R}\left(T_P\left(\boldsymbol{R}^{-1}(\boldsymbol{x})\right)\right)$$

$$\begin{tikzcd}
I \in \mathfrak{I}_P \arrow[r, "T_P"'] & I' \in \mathfrak{I}_P \arrow[d, "\boldsymbol{R}"'] \\
\boldsymbol{x} \in D_{\boldsymbol{f}} \arrow[u, "\boldsymbol{R}^{-1}"'] \arrow[r, dotted, "\boldsymbol{f}_P"] & \boldsymbol{x}' \in D_{\boldsymbol{f}}
\end{tikzcd}$$

Figure 2: Relations between T_P, $\boldsymbol{R}$, and $\boldsymbol{f}_P$

Figure 2 illustrates the relations between T_P, $\boldsymbol{R}$, and $\boldsymbol{f}_P$.

Now we have a representation of the semantics of a given logic program as m-dimensional function, which we can also view as m real-valued functions. In general however, its domain is rugged and arbitrary precision is required. In the next Section 3.2, we will approximate this function in a controlled way and simplify its domain. This approximation will then be the target for the connectionist systems we build.

3.2 Constructing Piecewise Constant Functions

In the following, we assume P to be a covered logic program and $(\|\cdot\|, \dim)$ a bijective multi-dimensional level mapping on $\mathcal{B}_P$ which is, along with its inverse, effectively computable. $\boldsymbol{R}$ and $\boldsymbol{f}_P$ denote the embeddings with base b as defined in Section 3.1.

3.2.1 Approximating one Application of T_P

We will construct a ground sub-program of P such that the associated embedded single-step operator approximates the original one up to a given component-wise accuracy $\boldsymbol{\epsilon} \in \mathbb{R}^m$. To this end, we determine a greatest relevant level for each dimension, ensuring that any set of atoms of greater level will have an embedded value less than the given accuracy for that dimension. In this way, we can guarantee that our approximation obeys the accuracy requirements.

Definition 3.9. For all $\boldsymbol{l} \in \mathbb{N}^m$, the set of *atoms of level less than or equal to* $\boldsymbol{l}$ is defined as

$$\mathcal{A}_{\boldsymbol{l}} := \left\{ A \in \mathcal{B}_P \middle| \|A\| \leq l_{\dim(A)} \right\}.$$

Furthermore, the *instance of P up to $\boldsymbol{l}$* is defined as

$$P_{\boldsymbol{l}} := \left\{ A \leftarrow \text{body} \in \mathcal{G}(P) \middle| A \in \mathcal{A}_{\boldsymbol{l}} \right\}.$$

Definition 3.10. For all $\boldsymbol{l} \in \mathbb{N}^m$, the *greatest relevant input levels* with respect to $\boldsymbol{l}$ are $\hat{\boldsymbol{l}}$ with

$$\hat{l}_j := \max \left\{ \|L\| \middle| \dim(L) = j \text{ and } L \text{ is a body literal of some clause in } P_{\boldsymbol{l}} \right\}.$$

From the definitions and the computability assumptions we made, it is clear that $\mathcal{A}_{\boldsymbol{l}}$ and $P_{\boldsymbol{l}}$ are finite and, along with $\hat{\boldsymbol{l}}$, effectively computable.

Example 3.11. *In our running example, with* $\boldsymbol{l} = (2,2)$ *we have:*

$$\begin{aligned}
\mathcal{A}_{\boldsymbol{l}} &= \{e(0), o(0), e(s(0)), o(s(0))\} \\
P_{\boldsymbol{l}} &= \left\{ \begin{array}{r} e(0). \\ o(0) \leftarrow \neg e(0). \\ e(s(0)) \leftarrow o(0). \\ o(s(0)) \leftarrow \neg e(s(0)). \end{array} \right\} \\
\hat{\boldsymbol{l}} &= (2,1)
\end{aligned}$$

Note: In principle, a complete set of greatest relevant input levels could be defined for each output dimension separately. In the above example, we would then get greatest relevant input levels $(0,1)$ for output dimension 1 and greatest relevant input levels $(2,0)$ for output dimension 2. For simplicity, however, we drop this optimisation and use the above definition of $\hat{\boldsymbol{l}}$ which boils down to taking the component-wise maximum of all these separate vectors.

The following lemma establishes a connection between the single-step operators of some instance of P as in Definition 3.9 and the original program P.

Lemma 3.12. For all $\boldsymbol{l}, \boldsymbol{k} \in \mathbb{N}^m$, $\boldsymbol{k} \geq \boldsymbol{l}$, and $I, J \in \mathcal{I}_P$, we have that $T_{P_{\boldsymbol{k}}}(I)$ and $T_P(J)$ agree on $\mathcal{A}_{\boldsymbol{l}}$ if I and J agree on $\mathcal{A}_{\hat{\boldsymbol{l}}}$, i.e.

$$I \cap \mathcal{A}_{\hat{\boldsymbol{l}}} = J \cap \mathcal{A}_{\hat{\boldsymbol{l}}} \quad \text{implies} \quad T_{P_{\boldsymbol{k}}}(I) \cap \mathcal{A}_{\boldsymbol{l}} = T_P(J) \cap \mathcal{A}_{\boldsymbol{l}}.$$

Proof. The claim follows directly from the facts that $P_{\boldsymbol{k}}$ contains all clauses of $\mathcal{G}(P)$ with heads from $\mathcal{A}_{\boldsymbol{l}}$, and that these clauses depend only on atoms from $\mathcal{A}_{\hat{\boldsymbol{l}}}$. □

Definition 3.13. The *greatest relevant output levels* with respect to some arbitrary $\boldsymbol{\epsilon} > \boldsymbol{0}$ are $\boldsymbol{o}^{(\boldsymbol{\epsilon})}$ with

$$\begin{aligned}
o_j^{(\boldsymbol{\epsilon})} &:= \min \left\{ n \in \mathbb{N} \middle| \sum_{\substack{\|A\|>n \\ \dim(A)=j}} R_j(A) < \epsilon_j \right\} \\
&= \min \left\{ n \in \mathbb{N} \middle| n > -\frac{\ln((b-1)\epsilon_j)}{\ln b} \right\}
\end{aligned}$$

The following theorem connects the embedded consequence operator of the subprogram $P_{\boldsymbol{o}^{(\boldsymbol{\epsilon})}}$ with the desired error bound $\boldsymbol{\epsilon}$. It constitutes the basis for later approximations using connectionist systems.

Theorem 3.14. For all $\boldsymbol{\epsilon} > \boldsymbol{0}$, we have that

$$\left| \boldsymbol{f}_P(\boldsymbol{x}) - \boldsymbol{f}_{P_{\boldsymbol{o}^{(\boldsymbol{\epsilon})}}}(\boldsymbol{x}) \right| < \boldsymbol{\epsilon} \quad \text{for all } \boldsymbol{x} \in D_{\boldsymbol{f}}.$$

Proof. Let $\boldsymbol{x} \in D_{\boldsymbol{f}}$ be given. From Lemma 3.12, we know that

$$T_{P_{\boldsymbol{o}^{(\boldsymbol{\epsilon})}}}(\boldsymbol{R}^{-1}(\boldsymbol{x})) = \boldsymbol{R}^{-1}(\boldsymbol{f}_{P_{\boldsymbol{o}^{(\boldsymbol{\epsilon})}}}(\boldsymbol{x}))$$

agrees with

$$T_P(\boldsymbol{R}^{-1}(\boldsymbol{x})) = \boldsymbol{R}^{-1}(\boldsymbol{f}_P(\boldsymbol{x}))$$

on all atoms A with

$$\|A\| \leq o^{(\boldsymbol{\epsilon})}_{\dim(A)}.$$

Thus, for each $j \in \{1, \ldots, m\}$ we have that the j-th components of $\boldsymbol{f}_{P_{o^{\boldsymbol{\epsilon}}}}(\boldsymbol{x})$ and $\boldsymbol{f}_P(\boldsymbol{x})$ agree on the first $o^{(\boldsymbol{\epsilon})}_j$ digits. The maximum deviation occurs if all later digits are 0 in one case and 1 in the other. In that case, the difference of the j-th components is

$$\sum_{\substack{\|A\|>n \\ \dim(A)=j}} R_j(A),$$

which is $< \epsilon_j$ by definition of $\boldsymbol{o}^{(\boldsymbol{\epsilon})}$. □

Example 3.15. *In our running example, for $\boldsymbol{\epsilon} = (0.1, 0.1)$ and $b = 3$ we have:*

$$\begin{aligned} o_1^{(0.1,0.1)} = o_2^{(0.1,0.1)} &= \min\left\{ n \in \mathbb{N} \middle| n > -\frac{\ln(2 \cdot 0.1)}{\ln 3} \right\} \\ &= \min\left\{ n \in \mathbb{N} \middle| n > 1.46 \right\} \\ &= 2 \end{aligned}$$

and therefore

$$\boldsymbol{o}^{(0.1,0.1)} = (2, 2)$$

Thus, $\boldsymbol{f}_{P_{(2,2)}}$ approximates $\boldsymbol{f}_P$ up to a maximum error of 0.1 in each dimension.

3.2.2 Iterating the Approximation

Now we know that one application of $\boldsymbol{f}_{P_{\boldsymbol{o}^{(\boldsymbol{\epsilon})}}}$ approximates $\boldsymbol{f}_P$ up to $\boldsymbol{\epsilon}$. But what will happen if we try to approximate several iterations of $\boldsymbol{f}_P$? In general, $\hat{\boldsymbol{o}}^{(\boldsymbol{\epsilon})}$ might be greater than $\boldsymbol{o}^{(\boldsymbol{\epsilon})}$, that is, the required input precision might be greater than the resulting output precision. In that case, we may lose precision with each iteration. So in order to achieve a given output precision after a certain number of steps, we increase our overall precision such that we can afford losing some of it. Since the precision might decrease with each step, we can only guarantee a certain precision for a given maximum number of iterations.

Theorem 3.16. For all $\boldsymbol{l} \in \mathbb{N}^m$ and $n \in \mathbb{N}$, we can effectively compute $\boldsymbol{l}^{(n)}$ such that for all $I \in \mathfrak{I}_P$, $\nu \leq n$, and $\boldsymbol{k} \geq \boldsymbol{l}^{(n)}$:

$$T^{\nu}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{\nu}_{P}(I) \text{ on } \mathcal{A}_{\boldsymbol{l}}.$$

Proof. By induction on n. Let $\boldsymbol{l} \in \mathbb{N}^m$ be given.

base $n = 0$**:** Obviously, $T^0_{P_{\boldsymbol{k}}}(I) = I = T^0_P(I)$. We set $\boldsymbol{l}^{(0)} := \boldsymbol{l}$.

step $n \rightsquigarrow n+1$**:** By induction hypothesis, we can find $\boldsymbol{l}^{(n)}$ such that for all $I \in \mathfrak{I}_P$, $\nu \leq n$, and $\boldsymbol{k} \geq \boldsymbol{l}^{(n)}$, $T^{\nu}_{P_{\boldsymbol{k}}}(I)$ agrees with $T^{\nu}_P(I)$ on $A_{\hat{\boldsymbol{l}}}$. With

$$\boldsymbol{l}^{(n+1)} := \max\{\boldsymbol{l}, \boldsymbol{l}^{(n)}\},$$

we then have for all $I \in \mathfrak{I}_P$, $\nu \leq n$, and $\boldsymbol{k} \geq \boldsymbol{l}^{(n+1)}$:

$$\begin{array}{lll} & T^{\nu}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{\nu}_P(I) \text{ on } A_{\hat{\boldsymbol{l}}} & (\boldsymbol{k} \geq \boldsymbol{l}^{(n)}) \\ \Rightarrow & T^{\nu+1}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{\nu+1}_P(I) \text{ on } \mathcal{A}_{\boldsymbol{l}} & (3.12) \end{array}$$

$T^0_{P_{\boldsymbol{k}}}(I) = I = T^0_P(I)$ completes the induction step.

□

This result may not seem completely satisfying. If we want to iterate our approximation, we have to know in advance how many steps we will need at most. Of course, we could choose a very large maximum number of iterations, but then the instance of P up to the corresponding level might become very large. But in the general case, we might not be interested in that many iterations anyway, since T_P does not necessarily converge.

For acyclic programs, however, T_P is guaranteed to converge, and given any $\boldsymbol{l} \in \mathbb{N}^m$, we can find $\boldsymbol{k} \in \mathbb{N}^m$ such that $T_{P_{\boldsymbol{k}}}$ is guaranteed to agree with T_P on $\mathcal{A}_{\boldsymbol{l}}$ after an unlimited number of applications.

Lemma 3.17. If P is acyclic, then for all $\boldsymbol{l} \in \mathbb{N}^m$ we can effectively compute $\boldsymbol{k} \in \mathbb{N}^m$ such that for all $n \in \mathbb{N}$ and $I \in \mathfrak{I}_P$,

$$T^{n}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{n}_{P}(I) \text{ on } \mathcal{A}_{\boldsymbol{l}}.$$

Proof. With $\boldsymbol{k} := (\max \boldsymbol{l}, \dots, \max \boldsymbol{l})$, acyclicity yields that $\boldsymbol{k} \geq \hat{\boldsymbol{k}}$. Let $I \in \mathfrak{I}_P$ be given. We prove by induction the stronger claim that for all $n \in \mathbb{N}$,

$$T^{n}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{n}_{P}(I) \text{ on } \mathcal{A}_{\boldsymbol{k}}$$

base $n = 0$**:** $T^0_{P_{\boldsymbol{k}}}(I) = I = T^0_P(I)$.

step $n \rightsquigarrow n+1$**:**

$$\begin{array}{lll} & T^{n}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{n}_P(I) \text{ on } A_{\boldsymbol{k}} & (\text{Induction Hypothesis}) \\ \Rightarrow & T^{n}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{n}_P(I) \text{ on } A_{\hat{\boldsymbol{k}}} & (\boldsymbol{k} \geq \hat{\boldsymbol{k}}) \\ \Rightarrow & T^{n+1}_{P_{\boldsymbol{k}}}(I) \text{ agrees with } T^{n+1}_P(I) \text{ on } \mathcal{A}_{\boldsymbol{k}} & (3.12) \end{array}$$

□

3.2.3 Simplifying the Domain

Now we have gathered all information and methods necessary to approximate $\boldsymbol{f}_P$ and iterations of it. It remains to simplify the domain of the approximation so that we can regard the approximation as a piecewise constant function. We do this by extending $D_{\boldsymbol{f}}$ to some larger set $D_{\boldsymbol{l}}$.

The idea is as follows. Since only input atoms A with $\|A\| \leq \hat{l}_{\dim(A)}$ play a role in $P_{\boldsymbol{l}}$, we have that all elements of $D_{\boldsymbol{f}}$ which differ only after the $\hat{l}_j$-th digit in each component $j \in \{1, \ldots, m\}$ are mapped to the same value by $\boldsymbol{f}_{P_{\boldsymbol{l}}}$. So we have hyper-intervals

$$H := \prod_{j=1}^{m} [x_j, y_j] \subseteq \mathbb{R}^m$$

such that all elements of $H \cap D_{\boldsymbol{f}}$ are mapped to the same value.

Example 3.18. *In our running example, one of these hyper-intervals is given by the following values for the endpoints, since we have* $\hat{\boldsymbol{l}} = (2, 1)$*:*

$$\boldsymbol{x} = (0.\underbrace{01}_{=}000\ldots_b\,, 0.\underbrace{0}_{=}000\ldots_b)$$
$$\boldsymbol{y} = (0.\overbrace{01}111\ldots_b\,, 0.\overbrace{0}111\ldots_b)$$

Obviously, in each dimension $j \in \{1, \ldots, m\}$ there are $2^{\hat{l}_j}$ disjoint such ranges $[x_j, y_j]$, yielding a total number of

$$\prod_{j=1}^{m} 2^{\hat{l}_j}$$

disjoint hyper-intervals. So we can extend $\boldsymbol{f}_{P_{\boldsymbol{l}}}$ to a function $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ which has a domain consisting of this number of disjoint hyper-intervals and is constant on each of them. We formalise these and some additional results in the following.

Definition 3.19. Given some $\boldsymbol{l} \in \mathbb{N}^m$, for each $j \in \{1, \ldots, m\}$ an ordered enumeration of all l_j-digit floating point numbers with base b can be computed as

$$d_{\boldsymbol{l},j,i} := \sum_{k=1}^{l_j} \left(\begin{cases} b^{-k} & \text{if } \left\lfloor \frac{i}{l_j - k + 1} \right\rfloor \mod 2 = 1 \\ 0 & \text{otherwise} \end{cases} \right) \qquad (0 \leq i < 2^{l_j}).$$

We define the interval length

$$\lambda_{\boldsymbol{l},j} := \sum_{\|A\| > l_j} R_j(A) = \frac{1}{(b-1) \cdot b^{l_j}} \qquad \text{for } 1 \leq j \leq m.$$

Now we define

$$\begin{aligned} D_{\boldsymbol{l},j,i} &:= [d_{\boldsymbol{l},j,i}, d_{\boldsymbol{l},j,i} + \lambda_{\boldsymbol{l},j}] && \text{for } 1 \leq j \leq m \text{ and } 0 \leq i < 2^{l_j} \\ D_{\boldsymbol{l},j} &:= \bigcup_{i=0}^{2^{l_j}-1} D_{\boldsymbol{l},j,i} && \text{for } 1 \leq j \leq m \\ D_{\boldsymbol{l},\boldsymbol{i}} &:= \prod_{j=1}^{m} D_{\boldsymbol{l},j,i_j} && \text{for } \mathbf{0} \leq \boldsymbol{i} < 2^{\boldsymbol{l}} \end{aligned}$$

and get two equivalent constructions for[4]

$$D_{\boldsymbol{l}} := \prod_{j=1}^{m} D_{\boldsymbol{l},j} = \bigcup_{\mathbf{0} \leq \boldsymbol{i} < 2^{\boldsymbol{l}}} D_{\boldsymbol{l},\boldsymbol{i}}.$$

Thus, each $D_{\boldsymbol{l},j}$ consists of 2^{l_j} pieces of equal length, or alternatively, each $D_{\boldsymbol{l},\boldsymbol{i}}$ is an m-dimensional hyper-interval. No matter which way we construct it, $D_{\boldsymbol{l}}$ consists of $\prod_{j=1}^{m} 2^{l_j}$ hyper-intervals, each of which has a size of

$$\prod_{j=1}^{m} \lambda_{\boldsymbol{l},j} = \prod_{j=1}^{m} \frac{1}{(b-1) \cdot b^{l_j}} = (b-1)^{-m} \prod_{j=1}^{m} b^{-l_j}$$

Example 3.20. *In Figure 3, the values of the $d_{\boldsymbol{l},j,i}$, $\lambda_{\boldsymbol{l},j}$, $D_{\boldsymbol{l},j,i}$, $D_{\boldsymbol{l},j}$, and $D_{\boldsymbol{l},\boldsymbol{i}}$ are computed explicitly for our running example.*

Lemma 3.21. For all $\boldsymbol{l} \in \mathbb{N}^m$, $D_{\boldsymbol{l}}$ is an extension of $D_{\boldsymbol{f}}$, i.e.

$$D_{\boldsymbol{l}} \supseteq D_{\boldsymbol{f}}$$

Proof. Let $\boldsymbol{l} \in \mathbb{N}^m$ and $\boldsymbol{x} \in D_{\boldsymbol{f}}$. Then for each $j \in \{1, \ldots, m\}$ there is an $i_j \in \{0, \ldots, 2^{l_j} - 1\}$ such that $d_{\boldsymbol{l},j,i_j}$ agrees with x_j on its l_j digits. But $D_{\boldsymbol{l},j,i_j}$ contains all numbers which agree with $d_{\boldsymbol{l},j,i_j}$ on its l_j digits, thus $\boldsymbol{x} \in D_{\boldsymbol{l},\boldsymbol{i}} \subseteq D_{\boldsymbol{l}}$. □

Lemma 3.22. For all $\boldsymbol{l} \in \mathbb{N}^m$ and $j \in \{1, \ldots, m\}$, the connected parts of $D_{\boldsymbol{l},j}$ are disjoint and the space between one part and the next is at least as wide as the parts themselves.

Proof. The minimum distance between two parts occurs when the left endpoints differ only in the last, i.e. l_j-th, digit. In that case, the distance between these endpoints is b^{-l_j}, which is $\geq 2 \cdot \lambda_{\boldsymbol{l},j}$ since $b \geq 3$. □

Lemma 3.23. For all $\boldsymbol{l} \in \mathbb{N}^m$ and $\mathbf{0} \leq \boldsymbol{i} < 2^{\hat{\boldsymbol{l}}}$, $\boldsymbol{f}_R$ is constant on $D_{\hat{\boldsymbol{l}},\boldsymbol{i}} \cap D_{\boldsymbol{f}}$.

[4]we use $\bigcup_{\boldsymbol{a} \leq \boldsymbol{i} < \boldsymbol{b}}$ for vectors instead of $\bigcup_{\boldsymbol{i}=\boldsymbol{a}}^{\boldsymbol{b}-\boldsymbol{1}}$ since it is not clearly defined how to count from one vector up to another

$$\begin{aligned}
d_{(2,1),1,0} &= 0.00_3 \quad d_{(2,1),1,1} = 0.01_3 & d_{(2,1),2,0} &= 0.0_3 \\
d_{(2,1),1,2} &= 0.10_3 \quad d_{(2,1),1,3} = 0.11_3 & d_{(2,1),2,1} &= 0.1_3 \\
\lambda_{(2,1),1} &= 0.0011\ldots_3 & \lambda_{(2,1),2} &= 0.011\ldots_3 \\
D_{(2,1),1,0} &= [0.00_3, 0.0011\ldots_3] & D_{(2,1),2,0} &= [0.0_3, 0.011\ldots_3] \\
D_{(2,1),1,1} &= [0.01_3, 0.0111\ldots_3] & D_{(2,1),2,1} &= [0.1_3, 0.111\ldots_3] \\
D_{(2,1),1,2} &= [0.10_3, 0.1011\ldots_3] & & \\
D_{(2,1),1,3} &= [0.11_3, 0.1111\ldots_3] & & \\
D_{(2,1),1} &= [0.00_3, 0.0011\ldots_3] & D_{(2,1),2} &= [0.0_3, 0.011\ldots_3] \\
&\cup [0.01_3, 0.0111\ldots_3] & &\cup [0.1_3, 0.111\ldots_3] \\
&\cup [0.10_3, 0.1011\ldots_3] & & \\
&\cup [0.11_3, 0.1111\ldots_3] & & \\
D_{(2,1),(0,0)} &= [0.00_3, 0.0011\ldots_3] & D_{(2,1),(0,1)} &= [0.00_3, 0.0011\ldots_3] \\
&\times [0.0_3, 0.011\ldots_3] & &\times [0.1_3, 0.111\ldots_3] \\
D_{(2,1),(1,0)} &= [0.01_3, 0.0111\ldots_3] & D_{(2,1),(1,1)} &= [0.01_3, 0.0111\ldots_3] \\
&\times [0.0_3, 0.011\ldots_3] & &\times [0.1_3, 0.111\ldots_3] \\
D_{(2,1),(2,0)} &= [0.10_3, 0.1011\ldots_3] & D_{(2,1),(2,1)} &= [0.10_3, 0.1011\ldots_3] \\
&\times [0.0_3, 0.011\ldots_3] & &\times [0.1_3, 0.111\ldots_3] \\
D_{(2,1),(3,0)} &= [0.11_3, 0.1111\ldots_3] & D_{(2,1),(3,1)} &= [0.11_3, 0.1111\ldots_3] \\
&\times [0.0_3, 0.011\ldots_3] & &\times [0.1_3, 0.111\ldots_3]
\end{aligned}$$

Figure 3: The values of the $d_{\boldsymbol{l},j,i}$, $\lambda_{\boldsymbol{l},j}$, $D_{\boldsymbol{l},j,i}$, $D_{\boldsymbol{l},j}$, and $D_{\boldsymbol{l},\boldsymbol{i}}$ for the running example.

$$\hat{\boldsymbol{f}}_{P_{(2,2)}}(\boldsymbol{x}) = \begin{cases}
(0.10_3, 0.11_3) = \boldsymbol{R}\left(\{e(0), o(0), o(s(0))\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(0,0)} \\
(0.11_3, 0.11_3) = \boldsymbol{R}\left(\{e(0), o(0), e(s(0)), o(s(0))\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(0,1)} \\
(0.10_3, 0.10_3) = \boldsymbol{R}\left(\{e(0), o(0)\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(1,0)} \\
(0.11_3, 0.10_3) = \boldsymbol{R}\left(\{e(0), o(0), e(s(0))\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(1,1)} \\
(0.10_3, 0.01_3) = \boldsymbol{R}\left(\{e(0), o(s(0))\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(2,0)} \\
(0.11_3, 0.01_3) = \boldsymbol{R}\left(\{e(0), e(s(0)), o(s(0))\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(2,1)} \\
(0.10_3, 0.00_3) = \boldsymbol{R}\left(\{e(0)\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(3,0)} \\
(0.11_3, 0.00_3) = \boldsymbol{R}\left(\{e(0), e(s(0))\}\right) & \text{for } \boldsymbol{x} \in D_{(2,1),(3,1)}
\end{cases}$$

Figure 4: The values of $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ for the running example.

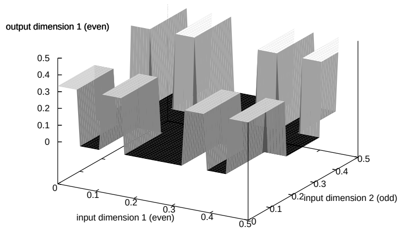
Proof. For all atoms A in bodies of clauses of $P_{\boldsymbol{l}}$, we have that $\|A\| \leq \hat{l}_{\dim(A)}$. Thus, in each dimension $j \in \{1, \ldots, m\}$, $T_{P_{\boldsymbol{l}}}$ regards only those atoms of level $\leq \hat{l}_j$, i.e. $T_{P_{\boldsymbol{l}}}$ is constant for all interpretations which agree on these atoms. This means that $\boldsymbol{f}_{P_{\boldsymbol{l}}}$ is constant for all $\boldsymbol{x}$ which agree on the first $\hat{l}_j$ digits of the jth component for all $j \in \{1, \ldots, m\}$, which holds for all $\boldsymbol{x} \in D_{\hat{\boldsymbol{l}},\boldsymbol{i}} \cap D_{\boldsymbol{f}}$. □

Definition 3.24. The *extension of $\boldsymbol{f}_{P_{\boldsymbol{l}}}$ to $D_{\hat{\boldsymbol{l}}}$*, $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}} : D_{\hat{\boldsymbol{l}}} \to D_{\boldsymbol{f}}$, is defined as

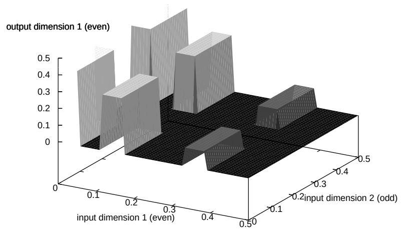
$$\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}(\boldsymbol{x}) := \boldsymbol{f}_{P_{\boldsymbol{l}}}(d_{\hat{\boldsymbol{l}},1,i_1}, \ldots, d_{\hat{\boldsymbol{l}},m,i_m}) \quad \text{for } \boldsymbol{x} \in D_{\hat{\boldsymbol{l}},\boldsymbol{i}}.$$

From the results above, it follows that $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ is well-defined.

Example 3.25. *The values of $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ for our running example are computed explicitly in Figure 4. The corresponding graphs are shown in Figure 5. Note also that, no matter where we start, iterated application of the function always ends up yielding $(0.10_3, 0.01_3) = \boldsymbol{R}(\{e(0), o(s(0))\})$, which is exactly the embedding of the fixed point of T_P restricted to $\mathcal{A}_{(2,2)}$.*



(a) output dimension 1



(b) output dimension 2

Figure 5: The graphs of $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ for the running example. The graphs of $\boldsymbol{f}_P$ would consist of dots slightly above the tops of these blocks.

We have simplified the domain of the approximated embedded single-step operator such that we can regard it as a function to $\mathbb{R}^m$ which is defined on a finite number of disjoint m-dimensional hyper-intervals, and which is constant on these.

Now, we will try to construct connectionist systems which either compute this function exactly or approximate it up to a given, arbitrarily small error vector. In the latter case we are facing the problem that the two errors might add up to an error which is greater than the desired maximum error. But this is easily taken care of by dividing the desired maximum overall error into one error $\boldsymbol{\epsilon}'$ for $\boldsymbol{f}_{P_{\boldsymbol{o}^{\boldsymbol{\epsilon}'}}}$ and another error $\boldsymbol{\epsilon}''$ for the connectionist system to be constructed.

4 Attempts for Sigmoidal Network Architectures

In the following, we will try to construct standard feed-forward networks with sigmoidal output functions in the hidden layer computing or approximating piecewise constant functions like the ones obtained in Section 3.2. We choose this architecture firstly because there exist well-established training algorithms, and secondly out of an intuition that a continuous network function will be better suited for dealing with the infinite nature of first order logic programs than discrete, propositional-like approaches. The reason for this intuition is that, in the context of the embedding we use and with infinite $\mathcal{B}_P$, the differences between embedded values can be arbitrarily small, falling below any fixed increment of some discrete function. Of course, on a real computer this argument is void, but intuitively we would like to stick to it at least in theory.

We will consider a function $g : D \to \mathbb{R}$ where $D \subset \mathbb{R}^m$ consists of a finite number of disjoint m-dimensional hyper-intervals. In each dimension $j \in \{1, \dots, m\}$, we have n_j edges of hyper-intervals, i.e. plain one-dimensional intervals, each of length b_j. On each of the hyper-intervals, g has a constant value. We will then compute or approximate this function g using connectionist systems. Each component of $\hat{\boldsymbol{f}}_{P_l}$ is a special case of such a function, so our results will be applicable to it. The simple intention with using g is to save some indices and get less complicated expressions.

So let $g : D \to \mathbb{R}$ be given by

$$D := \prod_{j=1}^{m} \bigcup_{i=0}^{n_j-1} [a_{j,i}, c_{j,i}],$$

or, equivalently,

$$D = \bigcup_{\boldsymbol{0} \le \boldsymbol{i} < \boldsymbol{n}} \prod_{j=1}^{m} [a_{j,i_j}, c_{j,i_j}]$$

with

$$g(\boldsymbol{x}) := y_{\boldsymbol{i}} \text{ for } \boldsymbol{x} \in \prod_{j=1}^{m} [a_{j,i_j}, c_{j,i_j}] \text{ and some } y_{\boldsymbol{i}} \in \mathbb{R},\, y_{\boldsymbol{i}} \ge 0,$$

where

$$c_{j,i} = a_{j,i} + b_j \text{ and } c_{j,i} < a_{j,i+1} \quad (1 \le j \le m, 0 \le i < n_j)$$

We want the output of the networks to agree with or to approximate g on D. We do not care about the output of the network for inputs outside D, since they are guaranteed not to be possible embeddings of interpretations, i.e. in our setting they do not carry any symbolic meaning which could be translated back to $\mathcal{I}_P$.

4.1 Intuitive Geometrical Attempt with 3 Layers

In the one-dimensional case examined in [Wit05, BHW05], we intuitively started the construction of the connectionist systems by positioning step functions between the constant pieces of the target function. If we want to extend this construction to multiple dimensions, we have to position the step functions accordingly in the multi-dimensional input space. Each step function divides the input space linearly and assigns a different output value to inputs from each of the two parts. Thus, once we have found a pattern to arrange the steps, it only remains to solve a set of linear equations to find out which outputs the step functions should use.

Figure 6 schematically shows two attempts to position the steps in an easy two-dimensional case. Attempt (a) aligns the steps along the input dimensions, putting one step between each row and column of the blocks representing the constant parts of the target function. If we use v_1 resp. v_2 to denote the output values that the step functions s_1 resp. s_2 assign to inputs with in_2 resp. in_1 values less than the step position, and w_1 resp. w_2 to denote the output values for inputs with greater in_2 resp. in_1 values, we obtain the following set of equations:

$$h_1 = v_1 + v_2 \qquad h_2 = v_1 + w_2$$
$$h_3 = w_1 + w_2 \qquad h_4 = w_1 + v_2$$

which resolves to the dependency

$$h_1 + h_3 = h_2 + h_4$$

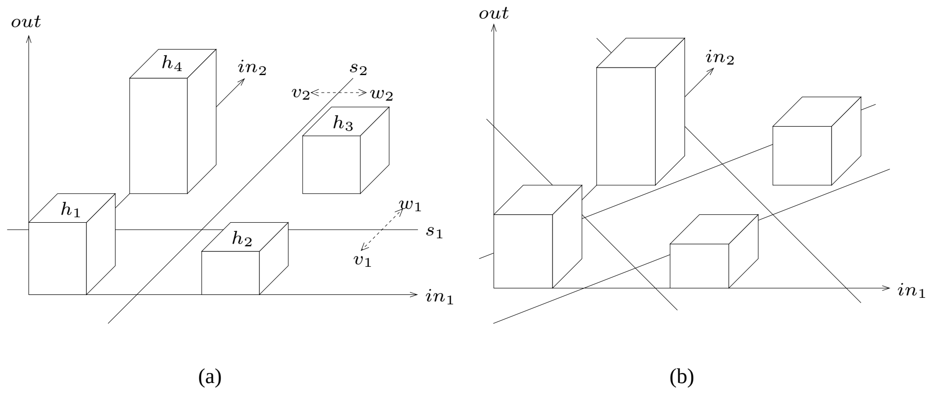


Figure 6: Schematic view of two attempts to position the step functions in a two-dimensional input space.

However, our target function g does not necessarily obey such restrictions. More specifically, for any given values h_1, h_2, h_3, h_4 we can construct a logic program and an approximation of it such that this combination of height values occurs in the embedded approximated single-step operator. Dependencies as the above one are thus not acceptable in the context of our application.

Attempt (b) works in this simple case, but fails with slightly more complicated target functions. Since all attempts to find intuitive patterns failed at some point, we abandoned this approach.

4.2 Discrete Construction with 4 Layers

We will now construct feed-forward networks with two hidden layers, each using weighted sum input functions. The networks have m input units whose values are fed to the units in the first hidden layer. Their outputs in turn serve as inputs for the units in the second hidden layer, whose outputs are summed up in the one output unit.

The construction works as follows: The first hidden layer is used to find out in which hyper-interval of D the input is contained. The second hidden layer has one unit for each hyper-interval, computing the value of g if the input is contained in the respective hyper-interval and 0 otherwise. Then we can sum up all results from the second hidden layer in the output unit to obtain the value of g.

We proceed in two steps. First we compute g exactly using a connectionist system with step activation functions in the hidden layers. Then we replace each step function by a corresponding sigmoidal function similar enough to guarantee that a given maximum error for the whole system is not exceeded.

4.2.1 Step Output Functions

In detail, in the first hidden layer for each dimension j and each interval $[a_{j,i}, c_{j,i}]$ we set up units whose summed output is 1 for inputs from this interval and 0 for inputs from the other intervals. In the second hidden layer, for each hyper-interval of D we sum up these values for the m intervals constituting the hyper-interval. If this sum equals m, we know that the input is contained in the hyper-interval. In this case, the respective unit in the second hidden layer should output the value of g on this hyper-interval, otherwise 0.

Each of the units in the two hidden layers has the following output function.

Definition 4.1 (Step function). For $x \in \mathbb{R}$,

$$s(x) := \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise.} \end{cases}$$

In the first hidden layer, for each dimension $j \in \{1, \ldots, m\}$ we need $n_j - 1$ units plus 2 "dummy" units to simplify the formalism. They can either be removed or replaced by a constant unit in a real implementation. The functions we intend to compute in the first hidden layer are defined in the following.

Definition 4.2 (Functions computed in the first hidden layer). For each input dimension $j \in \{1, \ldots, m\}$ and for each $i \in \{1, \ldots, n_j - 1\}$, we define

$$\theta_{j,i} := \tfrac{1}{2}(c_{j,i-1} + a_{j,i})$$

and, for all $x \in \mathbb{R}$,

$$\begin{aligned} s_{j,i}(x) &:= s(x - \theta_{j,i}) \\ s_{j,0}(x) &:= 1 \\ s_{j,n_j}(x) &:= 0 \end{aligned}$$

Lemma 4.3. For all $\boldsymbol{x} \in D$, $j \in \{1, \dots, m\}$ and $i \in \{0, \dots, n_j - 1\}$, we have that

$$s_{j,i}(x_j) - s_{j,i+1}(x_j) = \begin{cases} 1 & \text{if } x_j \in [a_{j,i}, c_{j,i}] \\ 0 & \text{otherwise} \end{cases}$$

Proof. The claim follows directly from Definitions 4.1 and 4.2. Figure 7 illustrates the corresponding functions for the first input dimension of our running example. Note that the claim is only for $\boldsymbol{x} \in D$, so input values that lie between the $D_{\boldsymbol{l},j,i}$ do not matter. □

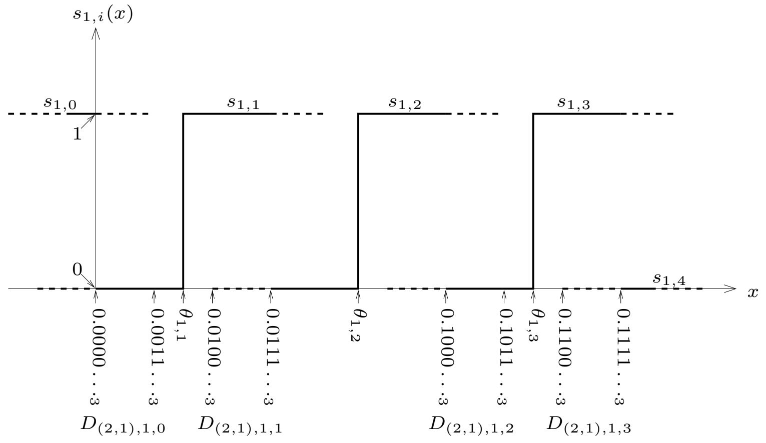


Figure 7: Illustration of the first hidden layer functions for the input dimension 1 of the running example.

Corollary 4.4. For all $\boldsymbol{x} \in D$ and $\boldsymbol{0} \leq \boldsymbol{i} < \boldsymbol{n}$, we have that

$$\sum_{j=1}^{m} \left(s_{j,i_j}(x_j) - s_{j,i_j+1}(x_j) \right) \begin{cases} = m & \text{if } \boldsymbol{x} \in \prod_{j=1}^{m} [a_{j,i_j}, c_{j,i_j}] \\ \leq m - 1 & \text{otherwise} \end{cases}$$

Proof. The claim follows from Lemma 4.3 and the fact that

$$\boldsymbol{x} \in \prod_{j=1}^{m} [a_{j,i_j}, c_{j,i_j}] \quad \text{iff} \quad x_j \in [a_{j,i_j}, c_{j,i_j}] \text{ for all } j \in \{1, \dots, m\}.$$

□

In the second hidden layer, we need one unit for each hyper-interval of D, yielding a total number of $\prod_{j=1}^{m} n_j$ units. The functions we want to compute in the second hidden layer are given in the following.

Definition 4.5 (Functions computed in the second hidden layer). For each $\boldsymbol{i} \in \mathbb{N}^m$ with $\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}$, we define $\theta_{\boldsymbol{i}} := m - 0.5$ and, for all $x \in \mathbb{R}$,

$$s'_{\boldsymbol{i}}(x) := s(x - \theta_{\boldsymbol{i}})$$

Lemma 4.6. For all $\boldsymbol{x} \in D$ and $\boldsymbol{i} \in \mathbb{N}^m$ with $\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}$, we have that

$$s'_{\boldsymbol{i}}\Big(\sum_{j=1}^{m} \big(s_{j,i_j}(x_j) - s_{j,i_j+1}(x_j)\big)\Big) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \prod_{j=1}^{m} [a_{j,i_j}, c_{j,i_j}] \\ 0 & \text{otherwise} \end{cases}$$

Proof. The claim follows from Definitions 4.1 and 4.5 and Corollary 4.4. □

Corollary 4.7. For all $\boldsymbol{x} \in D$, we have that

$$\sum_{\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}} y_{\boldsymbol{i}} \cdot s'_{\boldsymbol{i}}\Big(\sum_{j=1}^{m} \big(s_{j,i_j}(x_j) - s_{j,i_j+1}(x_j)\big)\Big) = g(\boldsymbol{x})$$

Proof. The claim follows from Lemma 4.6 and the fact that the hyper-intervals are disjoint. □

Now we have specified all functions we want to compute. It remains to define a network topology which implements these functions.

Definition 4.8 (The topology of the network).

$$\begin{aligned} V := \{&u_{\text{in},j} | 1 \leq j \leq m\} \cup \{u_{\text{out}}\} \cup \{u_{=1}\} \cup \\ &\{u_{j,i} | 1 \leq j \leq m, 0 \leq i \leq n_j\} \cup \\ &\{u_{\boldsymbol{i}} | \mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}\} \end{aligned}$$

is the set of input, output, constant 1, first hidden layer, and second hidden layer units, respectively. The hidden units and the output unit use the weighted sum input function. The hidden units use the step output function specified in Definition 4.1, while the output unit uses identity as output function.

An *edge* is a tuple $(\alpha, \beta, \gamma) \in V \times V \times \mathbb{R}$ with the intended meaning that the output of unit α is fed to unit β with weight γ. The set of edges is

$$\begin{aligned} E := &\bigcup_{j=1}^{m} \bigcup_{i=1}^{n_j - 1} \{(u_{\text{in},j}, u_{j,i}, 1), (u_{=1}, u_{j,i}, -\theta_{j,i})\} \cup \\ &\bigcup_{j=1}^{m} \big\{(u_{=1}, u_{j,0}, 1), (u_{=1}, u_{j,n_j}, 0)\big\} \\ &\bigcup_{\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}} \bigcup_{j=1}^{m} \big(\big\{(u_{j,i_j}, u_{\boldsymbol{i}}, 1), (u_{j,i_j+1}, u_{\boldsymbol{i}}, -1)\big\}\big) \cup \\ &\bigcup_{\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}} \{(u_{=1}, u_{\boldsymbol{i}}, -\theta_{\boldsymbol{i}}), (u_{\boldsymbol{i}}, u_{\text{out}}, y_{\boldsymbol{i}})\} \end{aligned}$$

with the values of $\theta_{j,i}$ and $\theta_{\boldsymbol{i}}$ as specified in Definitions 4.2 and 4.5.

Thus, we defined a network with m input, 1 output, 1 constant, $\sum_{j=1}^{m}(n_j+1)$ first hidden layer units (including the “dummy” units), and $\prod_{j=1}^{m} n_j$ second hidden layer units.

Example 4.9. *See Figure 8 to get an impression of the network resulting from our running example.*

Theorem 4.10. The function computed by the network from Definition 4.8 coincides with g on D.

Proof. With the specified network topology, the functions computed in the first and second layer correspond exactly to the functions defined in Definitions 4.2 and 4.5. The claim thus follows from Corollary 4.7. □

4.2.2 Sigmoidal Output Functions

We will now replace the step output functions by sigmoidal functions, keeping the topology from Definition 4.8.

Definition 4.11 (Sigmoidal function). For $x, z \in \mathbb{R}$,

$$\sigma_z(x) := \frac{1}{1+e^{-zx}}.$$

For all $x \in \mathbb{R} \setminus \{0\}$, we have $\lim_{z\to\infty} \sigma_z(x) = s(x)$. Since the value of the step function for input 0 was never relevant for us, we can approximate each step function by a sigmoidal function up to an arbitrarily small error in the relevant ranges. So we can approximate the whole step function network by a sigmoidal one up to any given error. It only remains to find out how large we have to choose the value of z for each unit.

In the following, for simplicity we will keep z as a parameter to the sigmoidal functions. In a real implementation, it can be incorporated in the connection weights.

Definition 4.12. Extending Definitions 4.2 and 4.5, for all $j \in \{1,\dots,m\}$ we define

$$\begin{aligned} \sigma_{j,i}(x) &:= \sigma_{z_{j,i}}(x-\theta_{j,i}) && (i \in \{1,\dots,n_j-1\}) \\ \sigma_{j,0}(x) &:= 1 \\ \sigma_{j,n_j}(x) &:= 0 \end{aligned}$$

and for each $\boldsymbol{i} \in \mathbb{N}^m$ with $\mathbf{0} \le \boldsymbol{i} < \boldsymbol{n}$, we define

$$\sigma'_{\boldsymbol{i}}(x) := \sigma_{z_{\boldsymbol{i}}}(x-\theta_{\boldsymbol{i}})$$

where $x \in \mathbb{R}$, and the values for the $z_{\boldsymbol{i}}$ and the $z_{i,j}$ are chosen according to Algorithm 1 with some given $\epsilon > 0$.

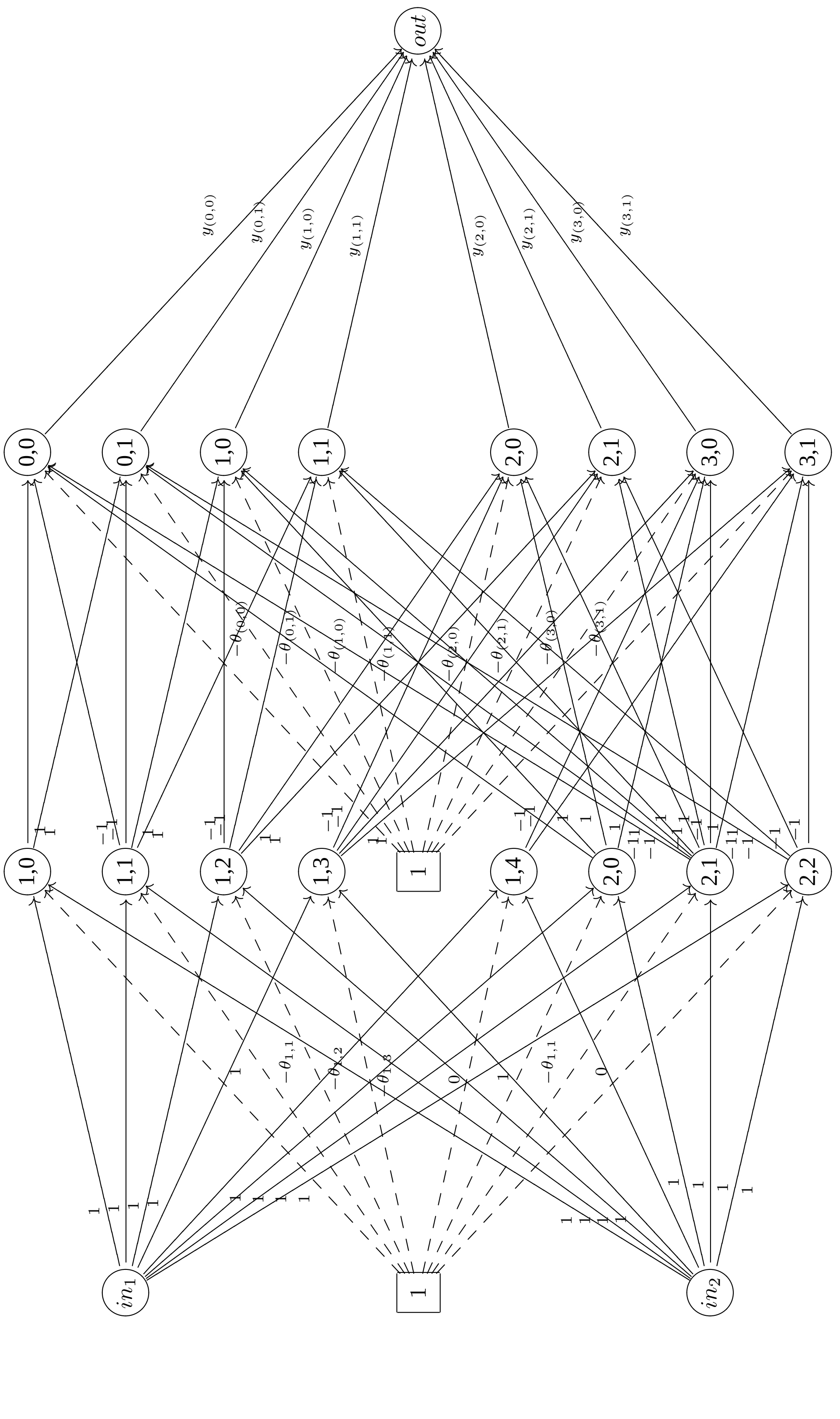


Figure 8: The network resulting from our running example according to Definition 4.8. The units in the first hidden layer are labelled with the values for (j, i), the units in the second hidden layer with those for $\boldsymbol{i}$. The θ labels and two occurrences of 1 and 0 belong to the connections drawn dashed for clarity. Note that the constant unit was duplicated in order to keep the figure less cluttered. Still it is only intended to convey a rough visual impression without the claim of being completely readable.

Algorithm 1: Computing $z_{\boldsymbol{i}}$ and $z_{j,i}$ from Definition 4.12

```
Input: Some $\epsilon > 0$
Output: Values for the $z_{\boldsymbol{i}}$ and the $z_{i,j}$
for $1 \leq j \leq m, 0 \leq i \leq n_j$ do
    $z_{i,j} \leftarrow 0$
end
$\epsilon' \leftarrow \frac{\epsilon}{(\prod_{j=1}^{m} n_j)-1}$
for $\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}$ do
    choose $\mu \in (m-1, m-0.5)$
    choose $z_{\boldsymbol{i}}$ such that $y_{\boldsymbol{i}} \cdot \sigma'_{\boldsymbol{i}}(\mu) < \epsilon'$
    choose $\nu \in (m-0.5, m)$ such that $y_{\boldsymbol{i}} \cdot \sigma'_{\boldsymbol{i}}(\nu) > y_{\boldsymbol{i}} - \epsilon$
    for $1 \leq j \leq m$ do
        increase $z_{j,i_j}$ such that $\sigma_{j,i_j}(a_{j,i_j}) \geq \frac{1}{2}\left(\frac{\nu}{m}+1\right)$
        increase $z_{j,i_j+1}$ such that $\sigma_{j,i_j+1}(c_{j,i_j}) \leq \frac{1}{2}\left(1-\frac{\nu}{m}\right)$
        increase $z_{j,i_j}$ such that $\sigma_{j,i_j}(c_{j,i_j-1}) \leq \mu-(m-1)$
        increase $z_{j,i_j+1}$ such that $\sigma_{j,i_j+1}(a_{j,i_j+1}) \geq 1-(\mu-(m-1))$
    end
end
```

Algorithm 1 obviously terminates and assigns values to all variables needed in Definition 4.12. It remains to prove that the algorithm is consistent, i.e. in all statements containing "such that" there actually exist values satisfying the condition, and that using the resulting assignments we really obtain an approximating network.

Lemma 4.13. Algorithm 1 is consistent.

Proof. For each of the critical statements, we give the reason why there is always a value satisfying the respective condition, using the facts that all $y_{\boldsymbol{i}} \geq 0$ as well as $\epsilon, \epsilon' > 0$, the property that $\lim_{z\to\infty} \sigma_z(x) = s(x)$ for all $x \in \mathbb{R} \setminus \{0\}$, and our knowledge about the $\theta_{j,i}$ and the $\theta_{\boldsymbol{i}}$.

line 7: $\mu < m - 0.5 = \theta_{\boldsymbol{i}}$

line 8: $m > m - 0.5 = \theta_{\boldsymbol{i}},\ |m-\theta_{\boldsymbol{i}}| \geq |\mu - \theta_{\boldsymbol{i}}|,\ \epsilon > \epsilon'$

line 10: $a_{j,i_j} > \theta_{j,i_j},\ \frac{1}{2}\left(\frac{\nu}{m}+1\right) < 1$

line 11: $c_{j,i_j} < \theta_{j,i_j+1},\ \frac{1}{2}\left(1-\frac{\nu}{m}\right) > 0$

line 12: $c_{j,i_j-1} < \theta_{j,i_j},\ \mu-(m-1) > 0$

line 13: $a_{j,i_j+1} > \theta_{j,i_j+1},\ 1-(\mu-(m-1)) < 1$

□

Theorem 4.14. For all $\epsilon > 0$, the function computed by the network from Definition 4.8 with the step functions replaced by the sigmoidal functions from Definition 4.12 approximates g up to ϵ on D.

Proof. All line numbers in this proof refer to Algorithm 1. Let $\boldsymbol{x} \in D$. For each $\boldsymbol{i}$ with $\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}$ and each dimension $j \in \{1, \dots, m\}$, we have that $\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j) \leq 1$ and

case 1 $x_j \in [a_{j,i_j}, c_{j,i_j}]$:

$$\begin{aligned} &\sigma_{j,i_j}(x_j) \geq \frac{1}{2}\left(\frac{\nu}{m} + 1\right) \wedge && \text{by line 10 and } x_j \geq a_{j,i_j} > \theta_{j,i_j} \\ &\sigma_{j,i_j+1}(x_j) \leq \frac{1}{2}\left(1 - \frac{\nu}{m}\right) && \text{by line 11 and } x_j \leq c_{j,i_j} < \theta_{j,i_j+1} \\ \Rightarrow\ &\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j) \geq \frac{\nu}{m} \end{aligned}$$

case 2 $x_j \notin [a_{j,i_j}, c_{j,i_j}]$:

$$\begin{aligned} &(i_j > 0 \wedge x_j \leq c_{j,i_j-1}) \vee && \\ &(i_j < n_j - 1 \wedge x_j \geq a_{j,i_j+1}) && \text{since } \boldsymbol{x} \in D \\ \Rightarrow\ &\sigma_{j,i_j}(x_j) \leq \mu - (m-1) \vee && \text{by line 12 if } x_j \leq c_{j,i_j-1} < \theta_{j,i_j} \\ &\sigma_{j,i_j+1}(x_j) \geq 1 - (\mu - (m-1)) && \text{by line 13 if } x_j \geq a_{j,i_j+1} > \theta_{j,i_j+1} \\ \Rightarrow\ &\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j) \leq \mu - (m-1) \end{aligned}$$

Thus, for each $\boldsymbol{i}$ with $\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}$ we have

case 1 $\boldsymbol{x} \in \prod_{j=1}^{m}[a_{j,i_j}, c_{j,i_j}]$:

$$\begin{aligned} &\sum_{j=1}^{m}\left(\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j)\right) \geq m \cdot \frac{\nu}{m} = \nu \\ \Rightarrow\ &y_{\boldsymbol{i}} \cdot \sigma'_{\boldsymbol{i}}\Big(\sum_{j=1}^{m}\left(\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j)\right)\Big) > y_{\boldsymbol{i}} - \epsilon \quad \text{by line 8 and } \nu > \theta_{\boldsymbol{i}} \end{aligned}$$

case 2 $\boldsymbol{x} \notin \prod_{j=1}^{m}[a_{j,i_j}, c_{j,i_j}]$:

$$\begin{aligned} &\text{there is } j \in \{1, \dots, m\} \text{ with } x_j \notin [a_{j,i_j}, c_{j,i_j}] \\ \Rightarrow\ &\sum_{j=1}^{m}\left(\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j)\right) \\ &\leq (m-1) + (\mu - (m-1)) = \mu \\ \Rightarrow\ &y_{\boldsymbol{i}} \cdot \sigma'_{\boldsymbol{i}}\Big(\sum_{j=1}^{m}\left(\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j)\right)\Big) < \epsilon' \qquad \text{by line 7 and } \mu < \theta_{\boldsymbol{i}} \end{aligned}$$

as well as $0 < y_{\boldsymbol{i}} \cdot \sigma'_{\boldsymbol{i}} < y_{\boldsymbol{i}}$ on $\mathbb{R}$.
Thus, for all $\boldsymbol{\iota}$ with $\mathbf{0} \leq \boldsymbol{\iota} < \boldsymbol{n}$, if $\boldsymbol{x} \in \prod_{j=1}^{m}[a_{j,\iota_j}, c_{j,\iota_j}]$ then

$$g(\boldsymbol{x}) = y_{\boldsymbol{\iota}}$$

and

$$\begin{aligned}
y_{\boldsymbol{\iota}} - \epsilon &< \sum_{\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}} y_{\boldsymbol{i}} \cdot \sigma'_{\boldsymbol{i}} \Big(\sum_{j=1}^{m} \big(\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j) \big) \Big) \\
&< y_{\boldsymbol{\iota}} + \Big((\textstyle\prod_{j=1}^{m} n_j) - 1 \Big) \epsilon' \\
&= y_{\boldsymbol{\iota}} + \Big((\textstyle\prod_{j=1}^{m} n_j) - 1 \Big) \frac{\epsilon}{(\prod_{j=1}^{m} n_j) - 1} && \text{by line 4} \\
&= y_{\boldsymbol{\iota}} + \epsilon
\end{aligned}$$

The fact that the sigmoidal version of the network from Definition 4.8 actually computes $\sum_{\mathbf{0} \leq \boldsymbol{i} < \boldsymbol{n}} y_{\boldsymbol{i}} \cdot \sigma'_{\boldsymbol{i}} \left(\sum_{j=1}^{m} \big(\sigma_{j,i_j}(x_j) - \sigma_{j,i_j+1}(x_j) \big) \right)$ follows analogously to the proof of Theorem 4.10. □

4.3 Conclusion: Failure

While the attempt in Section 4.1 failed right away, the construction presented in Section 4.2 is too clumsy and too fragile for a real implementation. There are many parameters that are related to each other, and if we change only some of them, the whole construction will break down. This means we cannot use standard training algorithms like backpropagation if we want to preserve the construction. Additionally, the construction with the counting first hidden layer is a very discrete approach, which is only disguised by the use of sigmoidal functions used to simulate local receptiveness. Having lost two main reasons for using sigmoidal feed-forward networks, we will therefore abandon this attempt.

5 Prerequisites for a Real Implementation

Freed from the requirement to use sigmoidal networks, we will now design specialised network architectures, accepting that we may also have to devise suitable training methods. According to the experiences from the previous section where we used sigmoidal networks just in order to simulate local receptiveness, we will now concentrate on locally receptive architectures right away.

Exploiting the domain knowledge we have will enable us to adapt or design networks architectures specifically tailored to our needs with only small numbers of free parameters. While in some cases standard training methods may still work, we will modify them or devise new methods to take advantage of these properties.

Unlike Section 4, this section will not deal with an arbitrary real-valued function defined on D. Firstly because we want to exploit our knowledge about the specific properties of the function we compute, and secondly because when implementing the resulting connectionist systems, we would like to avoid building a separate network for each output dimension.

As target function for the transformation algorithms, we will consider a function $\boldsymbol{g} : D_{\boldsymbol{l}} \to \mathbb{R}^m$ for some arbitrary $\boldsymbol{l} \in \mathbb{N}^m$ with $D_{\boldsymbol{l}}$ as defined in Definition 3.19. It corresponds to the extended embedded single step operator of a finite approximation of some logic program. Training data will be generated using (possibly approximated) logic programs. Whether or not $\boldsymbol{g}$ relates to the same logic program as the training data depends on the scenario.

Again, one could examine in each concrete case how much the input space granularity required by the individual output dimensions of $\boldsymbol{g}$ varies, and then decide whether to use a separate set of hidden layer units for each output dimension splitting the input space only as fine-grained as necessary for that output dimension, or to use one common set of hidden layer units and accept a possibly more fine-grained partition of the input space than necessary for some output dimensions. Since neither of these methods is in all cases the more efficient choice, we will stick to the second one and stay consistent with Definition 3.10 and the subsequent remark.

Since in a real implementation we are inherently dealing with finite interpretations, we could allow base 2 for the embedding without losing injectivity (see the remark after Definition 3.6). Considering the binary nature of computers, this is obviously an efficient choice, and the algorithms we will describe do not depend on the gaps between the hyper-intervals of D_l which disappear with base 2. However, to leave room for computational inaccuracy and to stay consistent with the theoretical background, we will stick to bases greater than 2.

In the following Section 5.1, we describe the general structure which all implemented network architectures have in common. Section 5.2 introduces the general training framework. In the remaining part of the thesis, we use this framework to present various network constructions in detail and discuss their properties.

5.1 Network Structure

All networks we discuss are 3-layer feed-forward networks with m units in the input layer and m units in the output layer, the latter computing the weighted sum of the hidden layer units' outputs. Each layer is fully connected to the next layer. This general structure is depicted in Figure 9.

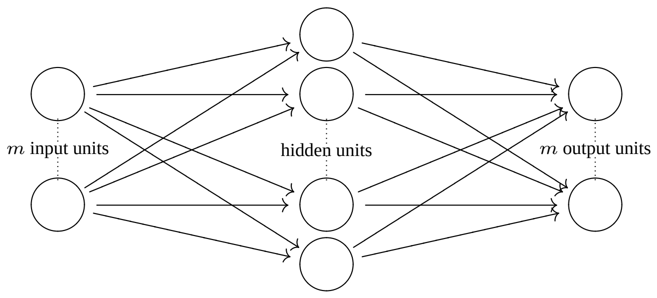


Figure 9: The general network structure used in the implementations.

5.1.1 Radial Basis Input Function

Each unit in the hidden layer has a radial basis input function computing the unit's activation, or potential, as defined in Algorithm 2. The ***weightsIn*** vector of a unit u (also denoted ***weightsIn***$_u$, but we will drop the subscript where it is clear from the context) contains the current weights of the incoming connections from the m input units, while its ***scaling*** vector (also denoted ***scaling***$_u$) is an optional internal parameter of u. The function $distance$ for a given network is the same in all hidden units. Usually it will be a metric, but this property is not required since we will not prove any results which might need it. In certain cases[5], a tolerant behaviour has proven useful to compensate for computational inaccuracy, moving inputs which are only slightly outside a unit's receptive area into the receptive area.

Algorithm 2: Radial Basis Input Function

```
    Input: Input in ∈ $\mathbb{R}^m$
   Output: Activation $act \in \mathbb{R}$
     Uses: weightsIn ∈ $\mathbb{R}^m$
           scaling ∈ $\mathbb{R}^m$
           $distance : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$
1 if scaling is used then
2     return $distance\,((\boldsymbol{in} - \boldsymbol{weightsIn}) * \boldsymbol{scaling}, \mathbf{0})$
3 else
4     return $distance(\boldsymbol{in}, \boldsymbol{weightsIn})$
5 end
```

5.1.2 Output Function

From the activation computed by the input function, the $outputFunction : \mathbb{R} \to \mathbb{R}$ produces the unit's final output. The output function depends on the network architecture, but it is the same for all hidden units and normally assigns greater output values to activations closer to 0.

The intuition is that the ***weightsIn*** vector of a unit u encodes a position in the input space. The closer a network input is to u's position, the lower u's activation becomes, thus the higher its output is. That is why this kind of units is also called *locally receptive*. The area in the input space where u's output is (significantly) different from 0 is called u's *receptive area* $rec(u)$.

The final output of a hidden unit u is fed forward to the output units via weighted connections. The weights of u's connections to all m output units form u's output weight vector ***weightsOut*** (also denoted ***weightsOut***$_u$). Each output unit computes the weighted sum of its inputs, i.e. it collects the inputs it receives from the hidden units multiplied by the corresponding weights and outputs the sum. Thus, a connection weight of 0 will disable the respective connection.

[5]see the paragraph about noise in Section 6.1.5

5.1.3 Winner Unit

The hidden unit that receives the least activation is called the *winner unit*. Its position in the input space is closest to the given input, and in the normal case described above it will be the unit with the highest output.

In some cases[6] we will use *winner-takes-all* behaviour: The winner unit is set to maximum output, all other units are set to output 0.

In other cases[7] we may need the slightly weaker *winners-share* behaviour: If the input lies in the receptive area of some (possibly several) units, nothing is changed. Only if this is not the case, the winner unit is set to maximum output.

5.2 The Training Framework

We train the networks by presenting positive examples, i.e. pairs of input vector and desired output vector. According to the context of this work, these pairs consist of an embedded interpretation and the embedded result of applying the consequence operator.

In some settings the training data may be *noisy*, i.e. we cannot be sure that each presented example is valid. The reason why we want to simulate such difficulties is that in the real world the training data may be obtained using sensors or other analog channels which are inherently noisy, or it may be modified by a malevolent third party. We will simulate two different kinds of noise:

Numeric Noise: The embedded interpretations have been modified numerically, usually in less significant digits. We will simulate this kind of noise by adding or subtracting (uniformly distributed) random values from some given range.

Semantic Noise: The example may encode valid interpretations, but it is semantically incorrect. The presented desired output is not a consequence of the presented input in the context of the single-step operator to be learned. We will simulate this kind of noise by presenting a certain ratio of examples obtained from a different program than the target program.

The training algorithms work *online*, i.e. each example encountered is processed immediately and the network is adjusted. The examples are not stored.

Algorithm 3 is the general training algorithm used for all networks. In the following we will describe in detail each of the components it uses and conclude this section with some remarks about termination of the training process.

5.2.1 Procedure $adjust$

The procedure $adjust$ takes the given example and adjusts the network's weights and parameters so that it better fits the example. How exactly this is done depends on the network architecture.

[6] see Section 6.3.1
[7] see Section 6.1.5

Algorithm 3: Online Training

```
      Input: Positive example $(\boldsymbol{in}, \boldsymbol{out}_{des})$
       Uses: Network $N$
             Procedure $adjust(\boldsymbol{in}, \boldsymbol{out}_{des})$
             Error Computer $EC : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$
             Error Information $Inf_{error}$
             Initialisation Information $Inf_{init}$
             Utility Information $Inf_{util}$
             Refinement Criterion $RC$
             Procedure $refine()$
1 $adjust(\boldsymbol{in}, \boldsymbol{out}_{des})$
2 $\boldsymbol{out}_{act} \leftarrow N(\boldsymbol{in})$
3 $error \leftarrow EC(\boldsymbol{out}_{act}, \boldsymbol{out}_{des})$
4 update $Inf_{error}$
5 update $Inf_{init}$
6 update $Inf_{util}$ and remove inutile units
7 if refinement necessary according to $RC$ then
8     $refine()$
```

5.2.2 Error Computer

The Error Computer is a function which takes the desired output vector from the given example along with the actual output vector computed by the network and computes a single error value in $\mathbb{R}$ which represents the "badness" of the error made. As for the $distance$ function in Section 5.1.1, we do not require the Error Computer to be a metric. In fact, the Error Computer used in Section 6.1.3 is not a metric.

While it would in principle be possible to simply use a distance function like the Euclidean distance, we will usually use the slightly more general and adaptable Error Computer given in Algorithm 4.

Algorithm 4: The general Error Computer

```
      Input: Actual output $\boldsymbol{out}_{act}$
             Desired output $\boldsymbol{out}_{des}$
     Output: Error value $\in \mathbb{R}$
       Uses: $deviation : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m$
             $errorNorm : \mathbb{R}^m \to \mathbb{R}$
 Parameters: $squareError \in \{true, false\}$
1 $\boldsymbol{dev} \leftarrow deviation(\boldsymbol{out}_{act}, \boldsymbol{out}_{des})$
2 $error \leftarrow errorNorm(\boldsymbol{dev})$
3 if $squareError$ then
4     $error \leftarrow error^2$
5 return $error$
```

It proceeds in three steps. First, a deviation vector is computed from the two input vectors. Then a norm is computed from this deviation vector. Finally, the resulting value is optionally squared. All three steps are customisable by plugging appropriate functions resp. values into the Error Computer; for the $deviation$ function, we will usually use the Deviation Computer given in Algorithm 5.

Algorithm 5: The standard Deviation Computer

```
     Input: Actual output $\boldsymbol{out}_{act}$
            Desired output $\boldsymbol{out}_{des}$
    Output: Deviation $\in \mathbb{R}^m$
      Uses: $differenceToDeviation : \mathbb{R} \to \mathbb{R}$
Parameters: Tolerance vector $\boldsymbol{\epsilon} \in \mathbb{R}^m$
            $discardDeviationLessThanEpsilon \in \{true, false\}$
            $computeDeviationRelativeToEpsilon \in \{true, false\}$

1 dev ← $differenceToDeviation(\boldsymbol{out}_{des} - \boldsymbol{out}_{act})$
2 if $discardDeviationLessThanEpsilon$ then
3     for $1 \le j \le m$ do
4         if $dev_j < \epsilon_j$ then $dev_j \leftarrow 0$
5 if $computeDeviationRelativeToEpsilon$ then
6     dev ← dev/$\boldsymbol{\epsilon}$
7 return dev
```

Here, further options are given. The absolute value function is the most obvious choice for $differenceToDeviation$, but we will also use other functions. There is also the possibility to specify a tolerance vector $\boldsymbol{\epsilon}$, and to discard deviation components less than the corresponding component of $\boldsymbol{\epsilon}$ or to compute the deviation relative to $\boldsymbol{\epsilon}$ in order to give more weight to deviations in dimensions with small tolerance.

5.2.3 Error Information

The Error Information stores the amount of error each individual unit has caused. With each presented example, the error computed by the Error Computer is ascribed to the unit that is considered to have caused it, which is usually the winner unit. Different ways of storing these errors can be thought of; however, in practice we always use the last one.

- Just store the *last error* for each unit.
- Store the sum and the number of errors caused by each unit and compute the *average error*.
- In each step, reduce all units' errors by a given factor $errorRed$ and add the new error for the unit that caused it, thus obtaining an *accumulated error*. Obviously, the average accumulated error decreases as the number of units increases, since each individual unit in average receives less examples. In cases where we consider the absolute error values, e.g. with threshold refinement as described in

Section 5.2.6, this is an undesirable effect which can impede the refinement process. A simple remedy consists in only reducing the error of the one unit where the new error is added. A drawback of this method is that the error information for units which have not received any inputs for a long period of time could be considered obsolete and cause inappropriate refinements. See the paragraphs about Error Information in Sections 6.1.3 and 6.3.3 for a discussion of this issue in practice.

5.2.4 Initialisation Information

The Initialisation Information is used when adding new units in the refinement step. While the concrete content of this information obviously depends on the network architecture, there are some general strategies for how to obtain it in case the receptive areas of potential new units are known and of finite number. For each potential new unit we may store and update the following information when we encounter examples whose input vectors lie in its receptive area:

- The component-wise *minimum*[8], *maximum*, or *average* desired output values *encountered*.
- A *pseudo-learned*[8,9] output vector, i.e. we apply the same adjustments as to actually existing units.

This information can then be used to initialise new units' output weights in a sensible way. Other possibilities to initialise new units include using some *constant value* like 0 or the maximum embedded interpretation[8], or *deducing*[10] information from existing units. In these cases, there is be no need to store any initialisation information.

5.2.5 Utility Information

The Utility Information is used to determine and remove inutile units, i.e. units which do not affect the outcome of the network to a significant extent. This is used to optimise the network in settings where units need to be inserted in the training process which later become unnecessary, where noisy data leads to spurious units, or where the network is initialised inappropriately. The concrete content of this information depends on the network architecture and the training method, but a general idea is to decrease utilities over time and to add the difference of the actual network output and the imaginary network output without the winner unit to that unit's utility. In this work, utility information is only used in Section 6.3.3.

5.2.6 Refinement Criterion

The Refinement Criterion decides when and which unit should be refined. The following refinement criteria can be used:

[8]see Section 6.1.3
[9]see Section 6.3.3
[10]see Section 6.2.4

- *Threshold refinement*: Refine units whose error exceeds a given threshold.
- *Periodical refinement*[11]: After a fixed or increasing number of examples, refine the unit with the greatest error.
- *Periodical threshold refinement*[12]: As above, but refine only if that error exceeds a given threshold.

5.2.7 Procedure $refine$

As with the procedure $adjust$, the procedure $refine$ depends on the concrete network structure and training method, but generally it involves adding hidden units.

5.2.8 Termination

The termination criterion depends on the scenario. If positive examples can be generated artificially, a network can be *pre-trained* to a given accuracy. Possible measures for the accuracy include the error caused by the network on the presented examples, i.e. the Error Information described above, or the error caused on a certain reference set which need not be a subset of the presented examples. The intention with the latter is to avoid over-adaptation to the training data by terminating the training when accuracy on the reference set ceases to increase. Additional indicators for termination may be the number of examples presented or the number of hidden units used by the network.

Since we are dealing with online training algorithms, it is possible for a running system to learn continuously from its observations. In such *always-on* scenarios, training never terminates but instead causes adequate adjustments each time a positive example is encountered. One interesting question to investigate here is whether a system stays stable over a longer period of time when presented with examples that do not require adjustments.

6 Implemented Network Architectures

In this section, we will develop or adapt several network architectures in the context and framework described in Section 5. Each architecture's properties will be discussed theoretically, and in Section 7 we will examine them using statistics from actual implementations.

6.1 Cuboid Networks

The Cuboid Network architecture corresponds directly to the function $\boldsymbol{g}$ we want to compute. In this way we hope to be able to optimally exploit our background knowledge and to minimise the number of parameters to be trained.

The receptive areas of the hidden layer units are hyper-intervals. If an input lies inside the receptive area of a certain unit, then the unit outputs 1, otherwise 0. The

[11] see Section 6.2.3

[12] see Sections 6.1.3 and 6.3.3

unit's contribution to the final output for inputs from its receptive area is encoded in its ***weightsOut*** vector, the weights of its connections to the m output units.

In the two-dimensional case, the graph of each output dimension of the network function can be depicted using cuboids, hence the name.

6.1.1 Definition of the Hidden Layer

In more detail, the hidden units use the Radial Basis Input Function from Algorithm 2 and the following parameters and output function:

- The ***weightsIn*** from the input units to the hidden units encode the centre of the hidden units' receptive areas, as described in Section 5.1.1.
- The ***scaling*** parameter is used to determine the size of the receptive area: the larger the value of ***scaling*** in some dimension, the smaller the receptive area along that dimension.
- The $distance$ function for all hidden units is set to the Maximum metric:

  $$(\boldsymbol{x}, \boldsymbol{y}) \mapsto \max |\boldsymbol{x} - \boldsymbol{y}|$$

  In this way we achieve rectangular receptive areas, in contrast to the elliptic receptive areas obtained with the Euclidean metric.
- As $outputFunction$ for all hidden units, we invert and shift the step function s from Section 4.2.1, obtaining:

  $$x \mapsto \begin{cases} 1 & \text{if } x \leq 1 \\ 0 & \text{otherwise.} \end{cases}$$

  This defines the receptive area of a hidden unit to be the area where the unit's activation is ≤ 1. The output for inputs from that area is 1 while it is 0 for other inputs.

All together we get units with rectangular receptive areas on which they output 1 and outside which they output 0, which is exactly what we wanted.

Definition 6.1 (Valid Cuboid Network). A Cuboid Network is *valid* if it satisfies the following three conditions:

(i) For any hidden unit u there must be $\boldsymbol{l} \in \mathbb{N}^m$ and $\boldsymbol{0} \leq \boldsymbol{i} < 2^{\boldsymbol{l}}$ such that u's receptive area $rec(u)$ coincides with $D_{\boldsymbol{l},\boldsymbol{i}}$ from Definition 3.19.

(ii) For any two hidden units with overlapping receptive areas $D_{\boldsymbol{l},\boldsymbol{i}}$ and $D_{\boldsymbol{l}',\boldsymbol{i}'}$, it must be the case that either $D_{\boldsymbol{l},\boldsymbol{i}} \subset D_{\boldsymbol{l}',\boldsymbol{i}'}$ or $D_{\boldsymbol{l}',\boldsymbol{i}'} \subset D_{\boldsymbol{l},\boldsymbol{i}}$.

(iii) The union of all hidden units' receptive areas must be a superset of $D_{\boldsymbol{f}}$.

These conditions ensure (i) that each hidden unit is responsible for some hyper-interval of the domain of a possible embedded approximated single-step operator, (ii) that a hierarchy among the hidden units can be deduced from their receptive areas, and (iii) that each valid embedded input interpretation lies in the receptive area of some hidden unit.

Definition 6.2. A unit u' with $rec(u') = H'$ is called a *descendant* of unit u with $rec(u) = H$ if $H' \subset H$. It is called a *child* if it is a descendant and there is no unit u'' with $rec(u'') = H''$ such that $H' \subset H'' \subset H$. In these cases, u is called *ancestor*, resp. *mother*, of u'.

6.1.2 Transformation

Algorithm 6 shows how to build a valid Cuboid Network computing the function $\boldsymbol{g}$.

Algorithm 6: The Transformation Algorithm into Cuboid Networks

```
     Input: The function $\boldsymbol{g} : D_{\boldsymbol{l}} \to \mathbb{R}^m$
    Output: A valid Cuboid Network $N$ computing $\boldsymbol{g}$
1 Initialise $N$ with $m$ input and output units and an empty hidden layer
2 foreach hyper-interval $H = \prod_{j=1}^{m} [a_j, c_j]$ of $D_{\boldsymbol{l}}$ do
3     Add a hidden unit $u$ to $N$ with
4         weightsIn$_u = \frac{1}{2}(\boldsymbol{a} + \boldsymbol{c})$
5         scaling$_u = \boldsymbol{1} / \left(\frac{1}{2}(\boldsymbol{c} - \boldsymbol{a})\right)$
6         weightsOut$_u = \boldsymbol{g}(\boldsymbol{x})$ for some $\boldsymbol{x} \in H$
7 end
8 return $N$
```

Theorem 6.3. The Cuboid Network N obtained from Algorithm 6 is valid, and the function it computes coincides with $\boldsymbol{g}$ on $D_{\boldsymbol{l}}$.

Proof. Consider the unit u of N which was inserted for a given hyper-interval $H = \prod_{j=1}^{m} [a_j, c_j]$ of $D_{\boldsymbol{l}}$. Remember that $\boldsymbol{g}$ is constant on each hyper-interval of $D_{\boldsymbol{l}}$. Now for each input vector $\boldsymbol{x} \in D_{\boldsymbol{l}}$ and dimension $j \in \{1, \ldots, m\}$ we have:

$$\begin{aligned} x_j \in [a_j, c_j] \quad & \text{iff} \quad x_j \geq a_j \text{ and } x_j \leq c_j \\ & \text{iff} \quad |x_j - \tfrac{1}{2}(a_j + c_j)| \leq \tfrac{1}{2}(c_j - a_j) \\ & \text{iff} \quad \left| \frac{x_j - \frac{1}{2}(a_j + c_j)}{\frac{1}{2}(c_j - a_j)} \right| \leq 1 \qquad\qquad \text{since } c_j - a_j > 0 \end{aligned}$$

According to Algorithms 2 and 6, u's input function will output:

$$\begin{aligned} act &:= distance\big((\boldsymbol{x} - \textit{\textbf{weightsIn}}_u) * \textit{\textbf{scaling}}_u, \boldsymbol{0}\big) \\ &= \max \left| \frac{\boldsymbol{x} - \frac{1}{2}(\boldsymbol{a} + \boldsymbol{c})}{\frac{1}{2}(\boldsymbol{c} - \boldsymbol{a})} \right| \end{aligned}$$

and with the observation made above we obtain

$$act \quad \begin{cases} \leq 1 & \text{if } \boldsymbol{x} \in H \\ > 1 & \text{otherwise.} \end{cases}$$

Thus, we have

$$outputFunction(act) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in H \\ 0 & \text{otherwise.} \end{cases}$$

Each of the m output units scales this value with the corresponding component of ***weightsOut***$_u$, thus u's contribution to N's output vector is

$$\begin{aligned} 1 \cdot \textbf{\textit{weightsOut}}_u &= \boldsymbol{g}(\boldsymbol{x}) && \text{if } \boldsymbol{x} \in H\text{, and} \\ 0 \cdot \textbf{\textit{weightsOut}}_u &= \mathbf{0} && \text{otherwise.} \end{aligned}$$

From the facts that Algorithm 6 inserts exactly one unit for each hyper-interval of $D_{\boldsymbol{l}}$ and that the hyper-intervals are disjoint, it follows that the network function coincides with $\boldsymbol{g}$ on D. Validity follows directly from

(i) Definition 3.19 because the hyper-intervals of $D_{\boldsymbol{l}}$ are exactly the $D_{\boldsymbol{l},\boldsymbol{i}}$

(ii) Lemma 3.22

(iii) Lemma 3.21.

$\square$

6.1.3 Training

The idea for training valid Cuboid Networks is as follows. As we encounter positive example pairs of input and desired output, we adjust the cuboids' heights, i.e. the ***weightsOut*** vectors. The ***weightsIn*** as well as the ***scaling*** vectors, which together determine the receptive areas, remain unchanged. Thus, validity is obviously not affected. Only when a refinement occurs, i.e. when some unit produces too much error and we have to insert new units, we will have to do changes to the receptive areas. But these will be done in a well-defined way, preserving the validity of the network. In this way, we introduce our knowledge of the particular domain into the training process.

Additionally, we impose two restrictions on the height adjustments to further confine the training process:

1. The cuboids' heights can only decrease. The intuition is that each cuboid should represent those atoms which are consequences of all interpretations embedded in its receptive area, and not a kind of average consequence. We hope this may make it easier to extract symbolic information from trained networks.

2. The cuboids' heights can only be positive. This is in line with the first restriction, as the cuboids represent consequences, not exceptions.

In the following, we describe how we set up each of the components of the training framework described in Section 5.2.

Procedure $adjust$ In principle, the adjustment strategy is, in each individual output dimension, to decrease the cuboid's height to the smallest desired output value encountered for inputs in its receptive area. We could, whenever we encounter a desired output value lower than the current height, simply set the height to that value. However, to be less susceptible to noise and to avoid pure "learning by heart", we will soften this adjustment by introducing a $learningRate$ parameter. Given the actual and the desired output vectors for some cuboid, the function described in Algorithm 7 will be used to compute the new, adjusted output vector. Note that this may result in network outputs that do not represent a valid embedded interpretation. If we want to use outputs as new inputs in an iteration, such cases should be taken care of as described for numeric noise in Section 6.1.5.

Algorithm 7: The $valueAdjustFunction$ for Cuboid Networks

```
     Input: Actual output vector $\boldsymbol{out}_{act}$
            Desired output vector $\boldsymbol{out}_{des}$
    Output: Adjusted output vector
Parameters: $learningRate \in [0, 1]$
1 return $learningRate \cdot \min\{\boldsymbol{out}_{des}, \boldsymbol{out}_{act}\} + (1 - learningRate) \cdot \boldsymbol{out}_{act}$
```

This function is used in Algorithm 8 to adjust the units involved in processing an example input ***in*** presented to the network. The idea is that the units involved in the computation are lowered towards the desired output (if it is lower than the actual output), proceeding from ancestor to descendant units. Only non-negative values are assigned to ***weightsOut*** vectors. Thus, both of the above restrictions on height adjustments are satisfied.

When a unit is lowered, its children are raised in order to preserve the overall output on their receptive areas. The last unit in this hierarchy is the unit with the smallest receptive area containing ***in***. It may still have children, but none of their receptive areas contain ***in***. Thus, lowering this unit causes the desired change in the computation for ***in***. This process is illustrated in Figure 10.

Error Computer We neglect negative errors, i.e. errors caused by desired output values less than the actual output. The reason is that, since the heights can only decrease, these errors will at some point of the process be handled by existing units, and new units are not allowed to have the necessary negative heights anyway. Therefore we only want positive errors to cause refinement by insertion of new units.

Additionally we assume that we are given a vector $\boldsymbol{\epsilon}$ specifying the tolerance for each output dimension. Errors less than this tolerance are neglected.

In the following we will define the specific settings needed to achieve the described behaviours in the training framework from Section 5.2. We use the general Error Computer from Algorithm 4 with the following parameters:

- $deviation$: standard Deviation Computer from Algorithm 5 with

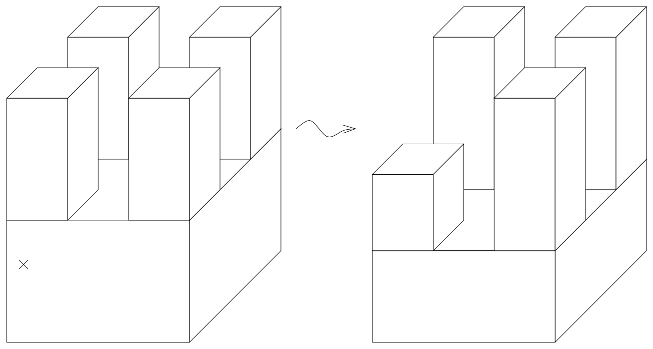


Figure 10: Illustration of the $adjust$ procedure for Cuboid Networks. The adjustment of one output dimension in a two-dimensional setting is shown schematically. The cross in the left part indicates the desired output.

Algorithm 8: The $adjust$ procedure for Cuboid Networks

```
Input: Input vector ***in***
       Desired output vector ***out***$_{des}$
Uses: Valid Cuboid Network $N$
      Output weight vectors ***weightsOut***
1  Find all hidden units $u_i$ ($0 \leq i < n$) of $N$ with
2      ***in*** $\in rec(u_i)$
3  and order them such that
4      $rec(u_0) \supset rec(u_1) \supset \cdots \supset rec(u_{n-1})$
5  for $0 \leq i < n$ do
6      ***new*** $\leftarrow \max\{\mathbf{0}, valueAdjustFunction($***weightsOut***$_{u_i}$, ***out***$_{des})\}$
7      foreach child $v$ of $u_i$ do
8          ***weightsOut***$_v \leftarrow$ ***weightsOut***$_v +$ ***weightsOut***$_{u_i} -$ ***new***
9      ***weightsOut***$_{u_i} \leftarrow$ ***new***
10     ***out***$_{des} \leftarrow$ ***out***$_{des} -$ ***new***
11 end
```

  - $differenceToDeviation$: $x \mapsto \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$
    We neglect negative errors as motivated above.
  - $\boldsymbol{\epsilon}$: as given
  - $discardDeviationLessThanEpsilon$: $true$
    We neglect errors within the given tolerance.
  - $computeDeviationRelativeToEpsilon$: $true$
    With this setting, we give more weight to errors occurring in an output dimension with lower tolerance.

- $errorNorm$: $\boldsymbol{x} \mapsto \max |\boldsymbol{x}|$
  Since we are given separate tolerance values for each output dimension, each single output dimension should be within this tolerance. Therefore we consider the maximum error rather than the Euclidean norm, for instance.

- $squareError$: $false$

Error Information All of the methods described in Section 5.2.3 for updating the units' errors work and make sense in certain settings. We use the *accumulated error*. The unit with the smallest receptive area still containing the input is considered to cause the error. Only that unit's error is reduced and increased, as described in Section 5.2.3. In practice, the issue mentioned there turns out not to be of great concern. Assuming a uniform distribution of the examples over time, the error accumulated in some unit can always be considered to be appropriate, since the unit's position in the input space and its receptive area do not change. Upon refinement, the error information is reset anyway, as defined in Section 6.1.4.

Initialisation Information The concrete information we use depends on the refinement method discussed in Section 6.1.4, but considering the architecture and the restrictions we imposed, three general ways of initialising new units make sense.

If we do not want to involve past examples, we can initialise new units such that the network's output in their receptive areas is the maximum embedded interpretation. Because of our restriction that the units' heights can only decrease, this is the only sensible *constant value* initialisation.

If we want to involve past examples, there are two options. In settings without noise, we can use the *minimum encountered* desired output, since the cuboid would have to shrink to that value anyway. However, the *pseudo-learned* initialisation will generally be preferable, enabling the network to deal with noise and avoiding over-adaptation, while retaining the advantage of exploiting knowledge about previously encountered examples.

Utility Information Two intuitive strategies of identifying inutile units can be thought of. Firstly, cuboids whose height in all dimensions has decreased to negligibly small values could be considered inutile. Secondly, neighbouring units of virtually equal height in all dimensions could be combined to one single unit. However, cases can be constructed where such seemingly inutile units are needed, e.g. to allow for further refinement of the respective area of the input space where some future descendant may have a height greater than zero. While methods are conceivable to deal with these issues, we did not implement utility measuring and unit removal mechanisms for this network architecture.

Refinement Criterion We use the *periodical threshold refinement* as described in Section 5.2.6 to avoid too hasty refinement as well as unnecessary refinement of units causing a low amount of error.

6.1.4 Refinement

For the $refine$ procedure in Algorithm 3, we consider two methods each having two variants, aware of the fact that there are many more methods and variants than can be covered here. The methods and variants are not designed to be changed during a training process.

Splitting Refinement A unit is refined by splitting its receptive area, inserting a new unit for each fragment, and removing the original unit. The split can occur (i) in one dimension of the input space, thus splitting the original unit into 2 new units, or (ii) in all dimensions at the same time, resulting in 2^m new units. If the receptive area of the unit u to be refined is $rec(u) = D_{\boldsymbol{l},\boldsymbol{i}}$ for some $\boldsymbol{l} \in \mathbb{N}^m$ and $\boldsymbol{0} \leq \boldsymbol{i} < 2^{\boldsymbol{l}}$, then

(i) the split occurs in a dimension $j \in \{1, \ldots, m\}$ and the new units' receptive areas are those two hyper-intervals of $D_{\boldsymbol{l}'}$ which are subsets of $D_{\boldsymbol{l},\boldsymbol{i}}$, where $\boldsymbol{l}' = (l_1, \ldots, l_j + 1, \ldots, l_m)$. The initialisation information consists of the new units' heights for each of the m possible split dimensions. If for some possible split dimension no examples have been encountered in the receptive area of some potential new unit, the according height information is set to the current unit's height at the time of the refinement.

    If $\boldsymbol{h}_j$ and $\boldsymbol{h}'_j$ are the heights of the new units resulting from a split in dimension j, then j is chosen such that $\frac{1}{2}\big(EC(\boldsymbol{h}_j, \boldsymbol{h}'_j) + EC(\boldsymbol{h}'_j, \boldsymbol{h}_j)\big)$ is maximal. The intuition behind this is that we want to choose the split dimension such that the difference between the new units is maximal with respect to the error computer. Since EC need not be symmetric, we take the average of both orders of application.

    In cases where the initial values do not provide useful information for determining the split dimension, e.g. with *constant value* initialisation, we simply split the dimension where $rec(u)$ has the greatest extent.

(ii) the new units' receptive areas are those 2^m hyper-intervals of $D_{\boldsymbol{l}'}$ which are subsets of $D_{\boldsymbol{l},\boldsymbol{i}}$, where $\boldsymbol{l}' = \boldsymbol{l} + \boldsymbol{1}$. The initialisation information consists of the new units' heights.

In both cases, the new units' error and initialisation information is initially empty.

Hierarchical Refinement A unit is refined by adding children. For a unit u with $rec(u) = D_{\boldsymbol{l},\boldsymbol{i}}$ for some $\boldsymbol{l} \in \mathbb{N}^m$ and $\boldsymbol{0} \leq \boldsymbol{i} < 2^{\boldsymbol{l}}$, we consider as possible children units that have as receptive area one of those hyper-intervals of $D_{\boldsymbol{l}'}$ which are subsets of $D_{\boldsymbol{l},\boldsymbol{i}}$, where $\boldsymbol{l}' = \boldsymbol{l} + \boldsymbol{1}$. Thus, for each unit there are 2^m possible children. When a child is added, its height in each respective dimension is set such that the network output for inputs from its receptive area is equal to the initialisation value, if possible; otherwise it is set to 0. Refinement can be done by

(i) adding one child at a time. The initialisation information contains the initial height for each of the 2^m possible children along with the sum of errors caused by examples whose input is contained in the possible child's receptive area. The child with the greatest sum is added.

(ii) adding all 2^m children in one step. The initialisation information only contains the initial height for all 2^m children.

In both cases, the child units' error and initialisation information is initially empty. The mother unit's error information is reset, and the initialisation information for the generated children is removed.

Comparison All methods and variants described above preserve validity of the Cuboid Network. The advantage with Splitting Refinement is that it results in less units. On the other hand, Hierarchical Refinement has the benefit that a unit can capture commonalities of interpretations embedded in adjacent parts of the input space, even if these commonalities consist of only some of the interpretations' consequences. Additional consequences belonging to a subset of these interpretations can then be produced by child units.

For both refinement methods, there is one variant increasing the number of units by 1 and another one increasing the number of units exponentially in the number of dimensions. The former option may take more refinement steps, while the latter option may add more unnecessary units.

6.1.5 Properties

Robustness We will discuss the semantic and structural effects of various kinds of damage that may affect a hidden layer unit u. We will not discuss damage to input and output units since they can simply be replaced. Generally it can be said that whenever damage affects the validity of the network, it will not recover from this damage.

- u fails: In general, we can only say that the output for inputs from $rec(u)$ decreases. In the case of a fairly trained network, however, where due to our training restrictions units describe the consequences common to all interpretations embedded in their receptive area, we can draw a semantic conclusion: The consequences captured by u will now simply be missing for inputs from $rec(u)$. This displays an advantage of Hierarchical Refinement, since here the consequences captured by ancestors or descendants of u are preserved. Additionally, if all possible children of u already exist, the structural effects of the failure can be coped with, although u's failure will be repaired with a possibly large number of previously unnecessary descendants. However, if not all possible children exist or if Splitting Refinement is used, there will be a hole in the input space not covered by any unit's receptive area, resulting in an output of $\mathbf{0}$. It is possible to detect and correct this during training, but care must be taken to avoid interference with the *winners-share* behaviour described below in the paragraph about noise.

- ***scaling*** corrupted: The size of $rec(u)$ is changed. The semantic effects for inputs which were previously contained in $rec(u)$ and now are outside correspond to those of failure described above. Inputs for which the opposite holds may get additional consequences, consequences may become erased, or even completely flipped, depending on the base of the embedding. In any case, the damage to the network's structure will not be repaired by the training methods and will also affect future refinements of u.

- ***weightsIn*** corrupted: The centre of $rec(u)$ is shifted. The effects correspond those of ***scaling*** corruption described above.

- ***weightsOut*** corrupted: u's height is changed, i.e. its contribution to the output for inputs from $rec(u)$. Because of the restriction that heights can only decrease in training, positive changes to ***weightsOut*** will simply be corrected by re-learning while negative changes can lead to previously unnecessary refinements.

Noise Noisy data can lead to wrong adjustment of units as well as unnecessary refinements. To a certain extent, the first effect is handled by using a $learningRate$ less than 1, while the second effect is avoided by considering multiple examples in the error computation, as with *average* or *accumulated error*. Still, due to the missing unit removal mechanism, noisy data may generally lead to an excessively growing number of hidden units.

Another effect can occur with numeric noise: If the corrupted input does not represent a valid embedded interpretation, it may not be contained in any unit's receptive area. In this case, the output will be $\mathbf{0}$ and no units will be adjusted. Due to the limited precision of floating point computations in a computer, these effects are likely to occur for inputs close to a receptive area's border, even if no noise has been added deliberately. To avoid them, the tolerant behaviour of the radial basis input function described in Section 5.1.1 or the *winners-share* behaviour described in Section 5.1.3 can be used.

Initialisation Issues In the following we will discuss three general ways of initialising a Cuboid Network:

- *Basic* initialisation yields a network to be trained from scratch. It consists of a unit that has a receptive area corresponding to the one hyper-interval of $D_\mathbf{0}$ and a height set to the maximum embedded interpretation.

- *Rough* initialisation, which corresponds to cases where the network is initialised with the transformation of some approximation $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ and then trained with embedded examples from the original program P or from a finer approximation $P_{\boldsymbol{l}'}$ with $\boldsymbol{l}, \boldsymbol{l}' \in \mathbb{N}^m$, $\boldsymbol{l}' \geq \boldsymbol{l}$.

  If for some unit u of the initial network, all training examples with $\boldsymbol{in} \in rec(u)$ have the same constant $\boldsymbol{out}_{des}$ value and $\boldsymbol{out}_{des}$ is strictly greater than u's output, then u may be refined unnecessarily, i.e. by units which all have the same height.

  If for some unit u of the initial network, all training examples with $\boldsymbol{in} \in rec(u)$ have a (not necessarily constant) $\boldsymbol{out}_{des}$ value strictly greater than u's output, then u may not satisfy the intention of capturing all consequences common to all interpretations embedded in $rec(u)$.

  Both of these undesirable properties can be avoided by initialising the network not with $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ but with

$$\begin{aligned}
&\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}} + \sum_{A \text{ with } \|A\| > l_{\dim(A)}} \boldsymbol{R}(A) \\
&= \hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}} + \sum_{\boldsymbol{k} \text{ with } \boldsymbol{k} > \boldsymbol{l}} b^{-\boldsymbol{k}} \\
&= \hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}} + \frac{1}{b-1}\left(\mathbf{1}/b^{\boldsymbol{l}}\right)
\end{aligned}$$

  i.e. by initially assuming all output atoms neglected by $\hat{\boldsymbol{f}}_{P_{\boldsymbol{l}}}$ to be true instead of false. It is easy to show that Theorem 3.14 still holds for this modified approximation.

  Rough initialisation is advantageous as compared to random or basic initialisation in that all consequences that hold in the initialisation function still hold in the training data, and thus the initially generated units would otherwise have to be learned during training. However, because of the simplifications we made in the definition of the hyper-intervals (e.g. uniformly partitioning the input space according to the greatest relevant input level, disregarding local possibilities of larger partitions), more units may be generated using rough initialisation than using training from basic initialisation.

- *Inappropriate* initialisation, i.e. initialisation using some function independent of the training examples.

  Training will still work, but since removal of units does not occur, the partitioning of the input space performed by the initial network can only be refined and not be freely restructured. Since the partitioning of the input space implicit in the training data may differ significantly from that of the initialisation function, this may lead to significant numbers of units which would not have been needed with some appropriate initialisation like rough or basic initialisation.

  Obviously, inappropriate initialisations can be tailored such that training performs worse than in the average case with random or basic initialisation in terms of runtime consumption and the number of units generated.

6.1.6 Wish List

Flexibility In an attempt to guarantee exploitation of the domain knowledge, the Cuboid Network architecture by design is very rigid. Most of the units' internal parameters are fixed or computed explicitly during training. While this has the advantage of a very small number of free parameters to be learned, intuitively we would like to soften this strict behaviour, achieving a more flexible architecture which may require less units or adapt better to concrete domain instances.

Robustness As described in Section 6.1.5, Cuboid Networks are quite susceptible to various kinds of damage or noise, and many of the resulting negative effects will not be remedied in the training process. We would like to have a more robust architecture which is able to recover from damage.

Unit Removal We would like to have a mechanism for detecting and removing units in order to reduce the number of unnecessary units and to counteract the unit producing effect of noise described in Section 6.1.5.

6.2 Supervised Growing Neural Gas

In the following we will review and adapt a network architecture called Supervised Growing Neural Gas (SGNG), as described in [Fri98]. In contrast to Cuboid Networks, SGNG is very flexible and general, with the drawback that it does not exploit the very special kind of functions we want to compute.

The hidden units' positions in the input space are not fixed, and SGNG uses a continuous approach without discrete and sharply delimited receptive areas. However, since the units' outputs quickly decrease towards 0 with increasing input distance, they can still be regarded as locally receptive.

Additionally, SGNG maintains a topology on the hidden units in order to determine which units should be considered neighbours and where units should be inserted or removed. This topology is adapted during training.

The idea is that SGNG expands in the input space just like gas would, and that more units will be located in areas where the output function is less smooth.

In the following, we will fit the SGNG architecture and training algorithm into our training framework from Section 5.2, slightly reordering the steps from the original algorithm. For a broader discussion of SGNG, see [Fri98].

6.2.1 Definition of the Hidden Layer

Definition 6.4. Let $\mathcal{U}$ be the set of hidden layer units of an SGNG N. Then N has a *topology* given by a set of undirected edges

$$\mathcal{E} \subseteq \big\{\{u_1, u_2\}\big|u_1, u_2 \in \mathcal{U}, u_1 \neq u_2\big\}$$

along with an *age* function

$$age : \mathcal{E} \to \mathbb{N}.$$

The *neighbours* of a unit $u \in \mathcal{U}$ are

$$\mathcal{N}(u) := \left\{u' \in \mathcal{U} \middle| \{u, u'\} \in \mathcal{E}\right\}.$$

The hidden units use the Radial Basis Input Function from Algorithm 2 and the following parameters and output function:

- The ***weightsIn*** from the input units to the hidden units encode the hidden units' positions in the input space.
- The ***scaling*** parameter is not used.
- As $distance$ function for all hidden units we use the Euclidean metric:

$$(\boldsymbol{x}, \boldsymbol{y}) \mapsto \sqrt{\sum_{j=1}^{m} (x_j - y_j)^2}$$

- The $outputFunction$ for all hidden units is the Gaussian Function

$$x \mapsto \exp\left(-\frac{1}{2}\frac{x^2}{\sigma^2}\right)$$

  The parameter σ determines the width of the function. While the function value rapidly decreases with increasing inputs, it never actually becomes 0. The larger σ, the wider the area in the input space becomes where the unit outputs values significantly above 0. To achieve a kind of local receptiveness where interference between units does not play a major role, σ of a unit u is always set proportional to u's average Euclidean distance from its topological neighbours $\mathcal{N}(u)$ in the input space.

However, since the functions we are dealing with are always non-negative, all hidden units will typically produce non-negative output (i.e. ***weightsOut*** will usually be $\geq \mathbf{0}$). This means that the interference will not be cancelled out but reinforced and may become significantly greater than 0, thus compromising the intention of locally receptive units. A remedy is to introduce a *bias unit*, i.e. a hidden layer unit which is, however, excluded from the topology and not connected to any input units. Instead it constantly outputs 1 and only its ***weightsOut*** are adjusted during training. No special treatment different from that for normal units is required for this adjustment. After some training, the bias unit's ***weightsOut*** will encode the average output. Thus, the normal units will have negative as well as positive ***weightsOut*** and the interference is more likely to be cancelled out.

6.2.2 Transformation

We did not devise a transformation algorithm for SGNG. The only way to obtain an SGNG approximating a given program is thus by training from scratch.

6.2.3 Training

In the following, we describe how each of the components of the training framework described in Section 5.2 is set up.

Procedure $adjust$ The adjustment procedure used by Algorithm 3 is defined in Algorithm 9 for SGNG.

Algorithm 9: The $adjust$ procedure for SGNG

```
     Input: Input vector in
            Desired output vector $\textbf{\textit{out}}_{des}$
      Uses: SGNG $N$ with units $\mathcal{U}$ and edges $\mathcal{E}$
            Input weight vectors weightsIn
            Output weight vectors weightsOut
            The output $out_u$ of any hidden unit $u$ in the last computation
Parameters: Winner position adaptation rate $\mu_w \in [0,1]$
            Neighbour position adaptation rate $\mu_n \in [0,1]$
            Output adaptation rate $\eta \in [0,1]$
            Maximum age for edges $a_{max} \in \mathbb{N}$

1  Determine winner unit $u_1$ and second winner $u_2$ for in
2  $\mathcal{E} \leftarrow \mathcal{E} \cup \big\{\{u_1, u_2\}\big\}$
3  $age\big(\{u_1, u_2\}\big) \leftarrow 0$
4  $\textbf{\textit{weightsIn}}_{u_1} \leftarrow \mu_w \cdot \textbf{\textit{in}} + (1-\mu_w) \cdot \textbf{\textit{weightsIn}}_{u_1}$
5  foreach neighbour $n \in \mathcal{N}(u_1)$ do
6      $\textbf{\textit{weightsIn}}_n \leftarrow \mu_n \cdot \textbf{\textit{in}} + (1-\mu_n) \cdot \textbf{\textit{weightsIn}}_n$
7  $\textbf{\textit{out}}_{act} \leftarrow N(\textbf{\textit{in}})$
8  foreach unit $u \in \mathcal{U}$ do
9      $\textbf{\textit{weightsOut}}_u \leftarrow \textbf{\textit{weightsOut}}_u + \eta \cdot out_u \cdot (\textbf{\textit{out}}_{des} - \textbf{\textit{out}}_{act})$
10 foreach edge $e = \{u_1, u'\} \in \mathcal{E}$ do
11     $age(e) \leftarrow age(e) + 1$
12     if $age(e) > a_{max}$ then
13         $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
14         if $\mathcal{N}(u') = \emptyset$ then $\mathcal{U} \leftarrow \mathcal{U} \setminus \{u'\}$
15     end
16 end
```

Error Computer As error measure we use the squared Euclidean distance, i.e. for EC in Algorithm 3 we use the function

$$(\textbf{\textit{out}}_{act}, \textbf{\textit{out}}_{des}) \mapsto \|\textbf{\textit{out}}_{act} - \textbf{\textit{out}}_{des}\|^2$$

where $\|\cdot\|$ denotes the Euclidean norm.

Error Information We use the *accumulated error* as described in Section 5.2.3, reducing all units' error in each step. The winner unit is considered to have caused the error.

Initialisation Information is not used.

Utility Information is not used.

Refinement Criterion We use the *periodical refinement* as described in Section 5.2.6.

6.2.4 Refinement

The refinement procedure for SGNG is given in Algorithm 10. Intuitively, the error information and the network topology are used to determine two units which might surround an area of the input space for which more fine-grained sampling is needed. Then a new unit is inserted with all parameters set to the average of the parent units' parameters.

Algorithm 10: The $refine$ procedure for SGNG

```
      Uses: SGNG $N$ with units $\mathcal{U}$
            Accumulated error $err(u)$ for any unit $u$
            Input weight vectors weightsIn
            Output weight vectors weightsOut
Parameters: Error reduction factor for parents $\alpha \in \mathbb{R}$

1  Determine unit $u_1 \in \mathcal{U}$ with maximum $err(u_1)$
2  Determine neighbour $u_2 \in \mathcal{N}(u_1)$ with maximum $err(u_2)$
3  $err(u_1) \leftarrow (1 - \alpha) \cdot err(u_1)$
4  $err(u_2) \leftarrow (1 - \alpha) \cdot err(u_2)$
5  Add a hidden unit $u$ to $\mathcal{U}$ with
6      $\boldsymbol{weightsIn}_u = \frac{1}{2}(\boldsymbol{weightsIn}_{u_1} + \boldsymbol{weightsIn}_{u_2})$
7      $\boldsymbol{weightsOut}_u = \frac{1}{2}(\boldsymbol{weightsOut}_{u_1} + \boldsymbol{weightsOut}_{u_2})$
8      $err(u) = \frac{1}{2}\big(err(u_1) + err(u_2)\big)$
9  $\mathcal{E} \leftarrow \mathcal{E} \setminus \big\{\{u_1, u_2\}\big\} \cup \big\{\{u_1, u\}, \{u, u_2\}\big\}$
10 $age(\{u_1, u\}) \leftarrow 0$
11 $age(\{u, u_2\}) \leftarrow 0$
```

6.2.5 Properties

Robustness As in Section 6.1.5, we consider the effects of damage to a hidden layer unit u. Because of the continuous and generic nature of SGNG, a semantic interpretation of the effects of damage for our specific area of application is generally not possible.

In general, any damage to u, the connection weights, or the SGNG topology can be coped with through further training. Failing units can be replaced and holes in the input space where no unit is responsible cannot occur due to the winner mechanism in the $adjust$ procedure. However, there is one caveat: If failure of a unit or damage to the SGNG topology causes a unit u to have $\mathcal{N}(u) = \emptyset$ and u is the maximum error unit, then refinement as defined in Algorithm 10 will fail. But over time u will either be winner (or second winner) for some input and thus be re-integrated into the topology, or otherwise its error will decrease such that at some point u no longer has the maximum error. In either case, refinement will work again.

Noise Like the effects of damage to the network discussed above, corruption in the training data is handled gracefully.

Initialisation Issues We did not devise an algorithm for transforming logic programs into SGNG networks, so there is only *basic initialisation*, generating an initial network with 2 units. While the output weights can be initialised randomly, the input weights are preferably initialised such that the units are positioned in opposite areas of $D_{\mathbf{0}}$. *Inappropriate initialisation* can be simulated by first training a network with examples from one program or randomly generated examples, and later switching to another program. Due to its flexible structure, SGNG is able to cope with this change.

6.2.6 Wish List

In the following we describe some desirable adaptations and inherent disadvantages of the general SGNG architecture in the context of our intended application. Due to its general nature, SGNG does not at all exploit the quite precise background knowledge we have about the input space, the approximate shape of the functions, or its semantics. We would expect to achieve more precise and faster learning results if we exploited this knowledge.

Output Function While hidden units with Gaussian output functions generally work, they are not very efficient for describing partially constant functions. Each constant piece has to be approximated by multiple Gaussian hidden units. Since all approximations we are dealing with are partially constant functions, and since even real examples from a non-approximated program can be computed by such functions within a given tolerance $\boldsymbol{\epsilon}$, we would prefer a more suitable kind of hidden units. For example, removing the Gaussian output functions and using the *winner-takes-all* behaviour described in Section 5.1.3 could drastically reduce the number of units needed.

Error Computer If we are given a per-dimension maximum tolerance $\boldsymbol{\epsilon}$, as we have always assumed, we would prefer to use the general Error Computer with settings as described in Section 6.1.3 for Cuboid Networks, with the exception that the function $differenceToDeviation$ should be $x \mapsto |x|$.

Refinement Criterion With a given tolerance ϵ, we would prefer to use *periodical threshold refinement* to avoid unnecessarily precise refinements.

Unit Removal Although there is a removal functionality built into the $adjust$ procedure, there are cases where it does not detect obviously unnecessary units, i.e. units whose removal does not affect the output significantly. They may have been necessary for earlier stages of the training or resulted from inappropriate initialisation. Two kinds of these unnecessary units are:

Dead units: Several neighbouring units may not be winners for any occurring input. Due to the nature of the input space $D_{\boldsymbol{f}}$, this can happen if units have been generated in an earlier stage of the training and later refinement has inserted surrounding units absorbing all inputs. The edges connecting the dead units to the rest of the network will eventually be removed by the $adjust$ procedure, but since none of these units ever receives any input, the edges between them will not age and not be removed, and thus neither the units themselves. We would like dead units to be removed.

Idempotent units: Multiple units may have been moved or generated in an area with constant or negligibly varying output and trained to have accordingly similar output weights. Since inputs are scattered in the whole area, the edges between these idempotent units are always renewed, thus the units will never be removed. With Gaussian output functions, these units are unavoidable to a certain extent, but they can occur even with the winner-takes-all behaviour described above. We would like to reduce the number of idempotent units, preferably keeping central units of the respective area, in order to ensure that the area is not taken over by other, unsuitable units.

Internal Topology While in general the topology learned by SGNG offers interesting information about the input space, it is not of much help in our application domain. We would like to get rid of the administrative overhead necessary for dealing with edges.

6.3 A Fine Blend

In this section we will present a network architecture which tries to combine the previous two approaches so as to fulfil the wish lists from Sections 6.1.6 and 6.2.6. One main point in this Fine Blend is that the domain knowledge is shifted from the fixed internal parameters of Cuboid Networks into the input metric. In this way we enable the hidden units to move around in the input space by adapting their ***weightsIn*** vectors, making the network more flexible and less susceptible to damage. Furthermore, we give up the attempt to incorporate some semantic intention into the output adjustment procedure by only allowing to decrease a unit's output. Additionally, we introduce a utility measure which enables us to remove units without the need of an internal topology.

In this way we keep the flexibility of SGNG while still specialising on our application domain and preserving the use of domain knowledge in the training process. A more complete comparison of properties is given in Table 1.

Network properties	**Cuboid Network**	**SGNG**	**Fine Blend**
Hidden units type	locally receptive	locally receptive	locally receptive (winner-takes-all)
Receptive area	discrete	continuous	discrete
Output function	discrete	continuous	discrete
Domain knowledge used	internal parameters, $distance$ function	no	$distance$ function
Transformation algorithm	yes	no	yes
Training properties			
Robustness (damage recovery)[a]	bad	good	good
Unit removal	no	using topology	using utility
Dead units[b]	–	to some extent	yes
Idempotent units[b]	–	no	yes
Internal topology	no	yes	no
Refinement Criterion	periodical threshold	periodical	periodical threshold
Domain knowledge used in			
Error Computer	yes	no	yes
output adjustment	yes	no	no
weightsIn adjustment	–	no	yes
refinement	yes	no	yes

Table 1: Properties of the three implemented network architectures.

[a]as described in Sections 6.1.5, 6.2.5, and 6.3.5
[b]as described in Section 6.2.6

We will need the notions of largest exclusive and smallest inclusive hyper-intervals. In order to avoid ambiguities, we first have to restrict the class of hyper-intervals we are talking about.

Definition 6.5. A hyper-interval $H \in \mathbb{R}^m$ is called *hyper-square of level* $l \in \mathbb{N}$ if $H = D_{\boldsymbol{l},\boldsymbol{i}}$ with $\boldsymbol{l} = (l, \ldots, l)$ and $\boldsymbol{0} \leq \boldsymbol{i} < 2^{\boldsymbol{l}}$. For precision restrictions in real implementations, we only define the hyper-squares up to some level $l_{max} \in \mathbb{N}$.

Now we can define the two notions mentioned above.

Definition 6.6. The *largest exclusive hyper-square* of a vector $\boldsymbol{u} \in D_{\boldsymbol{0}}$ and a set of vectors $V = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\} \subseteq \mathbb{R}^m$, 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 $V \cap H = \emptyset$.

An example for non-existing H_{ex} is illustrated in Figure 11.

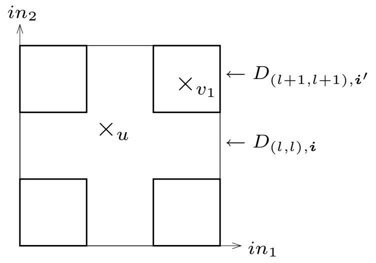

in_2
$\times_{v_1}$
$\leftarrow D_{(l+1,l+1),\boldsymbol{i}'}$
$\times_u$
$\leftarrow D_{(l,l),\boldsymbol{i}}$
in_1

Figure 11: Two-dimensional example where $H_{ex}(\boldsymbol{u}, \{\boldsymbol{v}_1\})$ does not exist.

Definition 6.7. The *smallest inclusive* (or *smallest common*) *hyper-square* of a non-empty set of vectors $U = \{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k\} \subseteq D_{\boldsymbol{0}}$, denoted by $H_{in}(U)$, is the hyper-square H of greatest level for which $U \subseteq H$.

In the normal case, both the inputs to the network and the ***weightsIn*** vectors of the hidden units are $\in D_{\boldsymbol{0}}$. If we have to deal with a corrupted vector $\notin D_{\boldsymbol{0}}$, we treat it equivalently to the euclidically closest vector in $D_{\boldsymbol{0}}$.

Algorithms 11 and 12 show how to compute H_{ex} and H_{in}.

6.3.1 Definition of the Hidden Layer

The hidden units use the Radial Basis Input Function from Algorithm 2 and the following parameters and output function:

- The ***weightsIn*** from the input units to the hidden units encode the hidden units' positions in the input space.
- The ***scaling*** parameter is not used.
- As $distance$ function for all hidden units we use the function defined in Algorithm 13. As mentioned above, corrupted vectors $\notin D_{\boldsymbol{0}}$ are treated equivalently to the euclidically closest vector $\in D_{\boldsymbol{0}}$.

Algorithm 11: Computing H_{ex}

```
      Input: $\boldsymbol{u} \in D_{\mathbf{0}}$
             $V \subseteq \mathbb{R}^m$
     Output: $H_{ex}(\boldsymbol{u}, V)$
 Parameters: Maximum level $l_{max} \in \mathbb{N}$

$l \leftarrow 0$
$H \leftarrow D_{\mathbf{0}}$
while $V \cap H \neq \emptyset$ and $l \leq l_{max}$ do
    if there is $\boldsymbol{i}$ with $\mathbf{0} \leq \boldsymbol{i} < 2^{(l+1,\ldots,l+1)}$ and $\boldsymbol{u} \in H' = D_{(l+1,\ldots,l+1),\boldsymbol{i}}$ then
        $l \leftarrow l + 1$
        $H \leftarrow H'$
    else
        return failure
    end
end
if $V \cap H = \emptyset$ then return hyper-square $H$ of level $l$
else return failure
```

Algorithm 12: Computing H_{in}

```
      Input: $U \subseteq D_{\mathbf{0}}$ with $U \neq \emptyset$
     Output: $H_{in}(U)$
 Parameters: Maximum level $l_{max} \in \mathbb{N}$

$l \leftarrow 0$
$H \leftarrow D_{\mathbf{0}}$
while $l \leq l_{max}$ do
    if there is $\boldsymbol{i}$ with $\mathbf{0} \leq \boldsymbol{i} < 2^{(l+1,\ldots,l+1)}$ and $U \subseteq H' = D_{(l+1,\ldots,l+1),\boldsymbol{i}}$ then
        $l \leftarrow l + 1$
        $H \leftarrow H'$
    else
        return hyper-square $H$ of level $l$
    end
end
return hyper-square $H$ of level $l$
```

The distance measure works as follows: Given some network input and some unit's position in the input space (encoded by its ***weightsIn*** vector), the first rough measure is their smallest common hyper-square. The greater its level, i.e. the smaller the hyper-square, the smaller the distance should be. This rough measure is then refined by a distinction of three cases[13], illustrated in Figure 12:

1. Both vectors also lie in (different) hyper-squares of greater level. In this case, their distance should be greatest among these cases.
2. One of the vectors lies in a hyper-square of greater level, while the other does not. In this case, their distance should be in the medium range among these cases.
3. None of the vectors lie in any deeper hyper-square. In this case, their distance should be smallest.

Finally, the Euclidean distance is used as a last refinement of the distance measure.

The intention with this measure is the following: A unit that is positioned in some hyper-square and not in any deeper one should be considered a default unit for that hyper-square. It should be responsible for inputs from that hyper-square, except for inputs from deeper hyper-squares containing another unit. A unit contained in some deeper hyper-square should only be responsible for inputs from higher hyper-squares if they do not have any default units and if the input is not from a different deeper hyper-square containing another unit. This behaviour is guaranteed by the case distinction without the need of directly comparing different units' positions.

The kinds of receptive areas resulting from this $distance$ function are illustrated in Figure 13.

The above case distinction could be refined by not only distinguishing whether the vectors lie in different deeper hyper-squares, but also how big the depth difference is. However, since the units' positions are not fixed and determined but expected to change, this measure is only meant as a rough estimation anyway, so we do not make this effort.

- The $outputFunction$ uses winner-takes-all behaviour: The unit with the least activation outputs 1, all other units output 0. Obviously this function cannot be computed by each unit separately. This may seem to impede parallelisability of the computation. However, since the output units have to wait for all hidden units to finish their computation anyway, this is not a major issue.

6.3.2 Transformation

Algorithm 14 shows how to build a Fine Blend computing a function which coincides with $\boldsymbol{g}$ on $D_{\boldsymbol{f}}$. Compared to Cuboid Networks, this is a restriction because we do

[13]note that with embedding base 2, only the first case would be left

Algorithm 13: The $distance$ function for the Fine Blend

```
Input: Inputs $\boldsymbol{x}, \boldsymbol{y} \in D_\mathbf{0}$
Output: Distance $\in \mathbb{R}$
if $\boldsymbol{x} = \boldsymbol{y}$ then return $0$
$H \leftarrow H_{in}(\{\boldsymbol{x}, \boldsymbol{y}\})$
$l \leftarrow$ level of $H$
if $H_{in}(\{\boldsymbol{x}\})$ and $H_{in}(\{\boldsymbol{y}\})$ are of greater level than $H_{in}(\{\boldsymbol{x}, \boldsymbol{y}\})$ then
    $malus \leftarrow \frac{2}{3}$
else if $H_{in}(\{\boldsymbol{x}\})$ or $H_{in}(\{\boldsymbol{y}\})$ is of greater level than $H_{in}(\{\boldsymbol{x}, \boldsymbol{y}\})$ then
    $malus \leftarrow \frac{1}{3}$
else
    $malus \leftarrow 0$
end
$euclDist \leftarrow$ Euclidean distance of $\boldsymbol{x}$ and $\boldsymbol{y}$
$maxDist \leftarrow$ diameter of $H$
return $1/\big(l + 1 - malus - 0.3 \cdot (euclDist/maxDist)\big)$
```

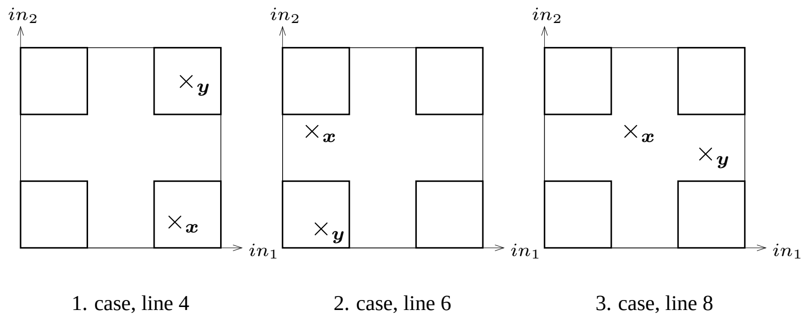
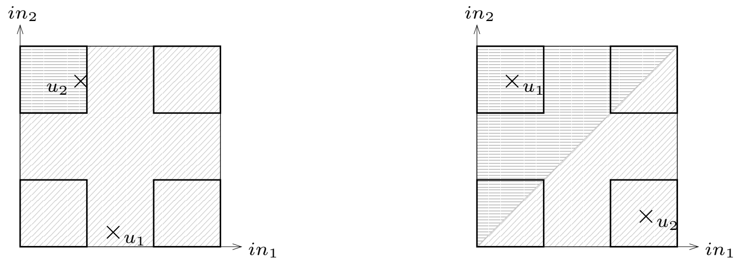


Figure 12: Illustration of the three cases in Algorithm 13



(a) u_1 is the default unit for the shown hyper-square; u_2 only wins for its own sub-hyper-square

(b) both u_1 and u_2 lie in different sub-hyper-squares of the shown hyper-square; only the Euclidean distance matters outside their sub-hyper-squares

Figure 13: Examples of receptive areas in Fine Blend.

not guarantee that the network function coincides with $\boldsymbol{g}$ on the whole domain of $\boldsymbol{g}$. This is due to the fact that since we dropped the use of the ***scaling*** parameter we can no longer scale the units' receptive area independently in each dimension. Thus, we consider a partition of the input space consisting only of hyper-squares, which possibly is a more fine-grained subset of the domain of $\boldsymbol{g}$. This is legitimate since the input space we consider still is a superset of $D_{\boldsymbol{f}}$, the set of all embedded interpretations. The number of units in the transformed network obviously is greater or equal to the number of units in a corresponding Cuboid Network, and in the worst case the increase is exponential in the number of dimensions; however, here we have the advantage that over time unnecessary units are detected and removed. In the early phase of training a transformed network, it is thus a good idea to use strict criteria for utility in order to faster get rid of the unnecessary units and soften them in the course of the training.

Algorithm 14: The Transformation Algorithm into a Fine Blend

```
Input: The function $\boldsymbol{g} : D_{\boldsymbol{l}} \to \mathbb{R}^m$
Output: A Fine Blend $N$ computing $\boldsymbol{g}$ for inputs from $D_{\boldsymbol{f}}$
1 Initialise $N$ with $m$ input and output units and an empty hidden layer
2 $l' \leftarrow \max \boldsymbol{l}$
3 foreach hyper-square $H = \prod_{j=1}^{m} [a_j, c_j]$ of level $l'$ do
4     Add a hidden unit $u$ to $N$ with
5         weightsIn$_u = \frac{1}{2}(\boldsymbol{a} + \boldsymbol{c})$
6         weightsOut$_u = \boldsymbol{g}(\boldsymbol{x})$ for some $\boldsymbol{x} \in H$
7 end
8 return $N$
```

Theorem 6.8. The network N obtained from Algorithm 14 coincides with $\boldsymbol{g}$ on $D_{\boldsymbol{f}}$.

Proof. Let $l' = \max \boldsymbol{l}$ as in the algorithm, and let $\boldsymbol{x} \in D_{\boldsymbol{f}}$. Then $\boldsymbol{x}$ is contained in exactly one hyper-square H of level l'. The algorithm positioned exactly one unit u inside H. Thus,

$$H_{in}(\{\boldsymbol{x}, \textbf{\textit{weightsIn}}_u\}) \geq \boldsymbol{l}',$$

and according to Algorithm 13, we have

$$distance(\boldsymbol{x}, \textbf{\textit{weightsIn}}_u) \leq \frac{1}{l' + 1 - \frac{2}{3} - 0.3} < \frac{1}{l'}.$$

Any other unit $u' \neq u$ is positioned in a different hyper-square, and since the hyper-squares are disjoint, we have that

$$l'' = H_{in}(\{\boldsymbol{x}, \textbf{\textit{weightsIn}}_{u'}\}) < l'$$

and therefore

$$distance(\boldsymbol{x}, \textbf{\textit{weightsIn}}_{u'}) \geq \frac{1}{l'' + 1} \geq \frac{1}{l'} \qquad \text{since } l', l'' \in \mathbb{N}.$$

Thus, u is the winner unit and its output 1 scaled by $\boldsymbol{weightsOut}_u$ becomes the output of the whole network. Because l' is defined to be the maximum of all levels in $\boldsymbol{l}$, H is contained in some hyper-interval of $D_{\boldsymbol{l}}$ and thus $\boldsymbol{g}$ is constant on H. Together with the definition of $\boldsymbol{weightsOut}_u$ in Algorithm 14, this yields the claim. □

6.3.3 Training

In the following, we describe how we set up each of the components of the training framework described in Section 5.2.

Procedure $adjust$ The adjustment procedure used by Algorithm 3 is defined in Algorithm 15 for the Fine Blend. The interesting point here is that the winner unit is not moved towards the input as with SGNG, but towards the centre of the smallest hyper-square including the unit and the input. This is in line with the definition of the $distance$ function and the intention that units should be positioned in the centre of the hyper-square for which they are responsible.

Algorithm 15: The $adjust$ procedure for the Fine Blend

```
     Input: Input vector $\boldsymbol{in}$
            Desired output vector $\boldsymbol{out}_{des}$
      Uses: Fine Blend $N$ with units $\mathcal{U}$
            Input weight vectors $\boldsymbol{weightsIn}$
            Output weight vectors $\boldsymbol{weightsOut}$
Parameters: Position adaptation rate $\mu \in [0,1]$
            Output adaptation rate $\eta \in [0,1]$
1 Determine winner unit $u$ for $\boldsymbol{in}$
2 $\boldsymbol{c} \leftarrow$ centre of $H_{in}(\{\boldsymbol{u}, \boldsymbol{in}\})$
3 $\boldsymbol{weightsIn}_u \leftarrow \mu \cdot \boldsymbol{c} + (1-\mu) \cdot \boldsymbol{weightsIn}_u$
4 $\boldsymbol{weightsOut}_u \leftarrow \eta \cdot \boldsymbol{out}_{des} + (1-\eta) \cdot \boldsymbol{weightsOut}_u$
```

Error Computer Assuming that we are given a tolerance vector $\boldsymbol{\epsilon} \in \mathbb{R}^m$ (otherwise the required changes are obvious), we use the general Error Computer from Algorithm 4 with the following parameters for the same reasons as in Section 6.1.3, where applicable:

- $deviation$: standard Deviation Computer from Algorithm 5 with
  - $differenceToDeviation$: $x \mapsto |x|$
  - $\boldsymbol{\epsilon}$: as given
  - $discardDeviationLessThanEpsilon$: $true$
  - $computeDeviationRelativeToEpsilon$: $true$
- $errorNorm$: $\boldsymbol{x} \mapsto \max |\boldsymbol{x}|$

- $squareError$: $false$

Error Information We use the *accumulated error* as described in Section 5.2.3, reducing and increasing only the winner unit's error. The issue that this may lead to obsolete error information is counteracted by the detection and removal of inutile units, which also include units that have not received any inputs for a long period of time, as described below in the paragraph about Utility Information.

Initialisation Information Each unit u maintains, for each dimension $j \in \{1, \ldots, m\}$ and each direction $d \in \{up, down\}$,

- an accumulated error $error_{j,d}(u)$ and
- a pseudo-learned[14] initialisation vector $\boldsymbol{init}_{j,d}(u)$

The idea is that each unit stores information about examples encountered in sub-hyper-squares of its receptive area, which will then be used in refinement to decide where a refining unit should be placed and what its initial output should be. In order to avoid a number of pieces of information exponential in the number of dimensions, and since with moving units this approach can only be a rough estimation anyway, we distinguish only for each dimension separately whether the example input was above or below the unit's position.

Concretely, all $error$ values are initialised with 0 and all $\boldsymbol{init}$ vectors with the respective unit's output, and with each example presented to the network, Algorithm 16 is invoked.

Cases are conceivable where this may lead to cyclic behaviour, i.e. where due to the imprecise and ambiguous error information a refining unit will repeatedly be placed in a position where it is not necessary and removed again for being inutile. This can be remedied by using 2^m $error$ values, one for each element of $\{up, down\}^m$. However, during our tests we did not encounter this problem.

Utility Information Each unit u maintains a utility value $utility(u)$, initially 1, which decreases over time and increases only when the unit makes a significant contribution to the network's output. A winner unit's contribution is determined by computing the difference between the error made by a hypothetical network without the winner unit and the network's actual error. If this difference is negative it is discarded, since the winner unit is then considered still to be in the adjustment phase and utility reduction is taken care of already. Otherwise, it is added to the winner unit's utility. If a unit's utility drops below a given threshold $utilityThreshold$, the unit is considered inutile and will be removed (unless it is the only remaining unit).

In certain cases this method can cause all idempotent units to be removed at the same time, while we obviously would like to retain at least one of them. For example, when for some reason an unnecessary refinement is performed in which two idempotent units are generated, their utilities will reach the threshold simultaneously. This is counteracted by a small random variation in the initial utility value.

[14] as described in Section 5.2.4

Algorithm 16: The Inf_{init} update procedure for the Fine Blend

```
Input: Input vector $\boldsymbol{in}$
       Desired output vector $\boldsymbol{out}_{des}$
Uses: Fine Blend $N$
      Error Computer $EC : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$
      Input weight vectors $\boldsymbol{weightsIn}$
      $error_{j,d}$ values and $\boldsymbol{init}_{j,d}$ vectors (Initialisation Information)
Parameters: Output adaptation rate $\eta \in [0, 1]$
            Error reduction factor $errorRed$
1  Determine winner unit $u$ for $\boldsymbol{in}$
2  $\boldsymbol{p} \leftarrow \boldsymbol{weightsIn}_u$
3  foreach dimension $j \in \{1, \ldots, m\}$ do
4      $error_{j,up}(u) \leftarrow (1 - errorRed) \cdot error_{j,up}(u)$
5      $error_{j,down}(u) \leftarrow (1 - errorRed) \cdot error_{j,down}(u)$
6      if $in_j < p_j$ then
7          $\boldsymbol{init}_{j,down} \leftarrow \eta \cdot \boldsymbol{out}_{des} + (1 - \eta) \cdot \boldsymbol{init}_{j,down}(u)$
8          $error_{j,down}(u) \leftarrow error_{j,down}(u) + EC(N(\boldsymbol{in}), \boldsymbol{out}_{des})$
9      else
10         $\boldsymbol{init}_{j,up} \leftarrow \eta \cdot \boldsymbol{out}_{des} + (1 - \eta) \cdot \boldsymbol{init}_{j,up}(u)$
11         $error_{j,up}(u) \leftarrow error_{j,up}(u) + EC(N(\boldsymbol{in}), \boldsymbol{out}_{des})$
12     end
13 end
```

In average, the more units a network has, the less often each unit wins and is able to make a contribution. Thus, using a constant utility reduction factor would decrease the units' average lifetime as the number of units increases. While this effect may be useful to restrict the size of a network, other, more direct and controlled means to achieve this should be used if necessary. We will therefore compensate for this effect by fixing a number of examples for which a unit should be allowed to be the winner without making a significant contribution. In each application of the Inf_{util} update procedure, we multiply this relative number of examples with the current number of units to obtain an absolute number of examples. Then we compute a utility factor which would decrease the initial utility of 1 to $utilityThreshold$ after this absolute number of examples has been presented to the network.

The whole utility information update procedure is given in Algorithm 17. Note that the output of the network without the winner unit can easily be obtained by determining the output of the second winner in the original network, i.e. the unit with the second least activation.

```
Algorithm 17: The $Inf_{util}$ update procedure for the Fine Blend
      Input: Input vector $\boldsymbol{in}$
             Desired output vector $\boldsymbol{out}_{des}$
       Uses: Fine Blend $N$ with units $\mathcal{U}$
             Error Computer $EC : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$
             $utility(u)$ for any unit $u$ (Utility Information)
 Parameters: Relative number $k \in \mathbb{R}^+$ of examples to survive
             $utilityThreshold \in [0, 1]$
1 $k' \leftarrow k \cdot |\mathcal{U}|$
2 $\phi \leftarrow \sqrt[k']{utilityThreshold}$
3 foreach unit $u' \in \mathcal{U}$ do $utility(u') \leftarrow \phi \cdot utility(u')$
4 Determine winner unit $u$ for $\boldsymbol{in}$
5 Let $N'$ be $N$ without unit $u$, or $\boldsymbol{x} \mapsto \mathbf{0}$ if $|\mathcal{U}| = 1$
6 $c \leftarrow EC(N'(\boldsymbol{in}), \boldsymbol{out}_{des}) - EC(N(\boldsymbol{in}), \boldsymbol{out}_{des})$
7 if $c > 0$ then $utility(u) \leftarrow utility(u) + c$
```

Refinement Criterion As with Cuboid Networks, we use the *periodical threshold refinement* described in Section 5.2.6.

6.3.4 Refinement

While the ***weightsIn*** adjustment facilitates 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, refinement should take care of densifying the network in areas where a great error is caused. Therefore, when a unit u is selected for refinement, we try to figure out the area it is responsible for and a suitable position to add a new unit.

If u 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 that still contains u. Now u is moved to the centre of this area, and the $error$ information gathered by u is used to determine a sub-hyper-square in whose centre a new unit is placed. The average of all ***init*** vectors which might stem from this sub-hyper-square is used to initialise the output of the new unit.

The exact procedure is specified in Algorithm 18.

6.3.5 Properties

Robustness As in Sections 6.1.5 and 6.2.5, we consider the effects of damage to a hidden unit u. Similar to SGNG, any damage to u or the connection weights can be repaired by further training, since all of these values are learned anyway. Failing units can be replaced, and holes in the input space as with Cuboid Networks cannot occur due to the winner-takes-all behaviour. The effects of damage are quite obvious:

- u fails: u's receptive area is taken over by other units, thus the specific results learned for inputs from u's receptive area are lost.
- ***weightsIn*** corrupted: u's position in the input space is changed. While this may cause no changes at all in the network function, generally it can alter u's receptive area, thus assigning different outputs to inputs that were contained in $rec(u)$ before and are no longer, and vice versa.
- ***weightsOut*** corrupted: u's output is changed.

Noise Noise is generally handled gracefully, since unnecessary adjustments or refinements can be undone in the further training process.

Initialisation Issues We consider the three ways of initialising a network as introduced in Section 6.1.5.

- *Basic* initialisation yields a network to be trained from scratch, consisting of one hidden unit positioned in the centre of the input space with an output of half the maximum embedded interpretation.
- *Rough* initialisation has the advantage that the error produced by the network will be smaller in the early phase of training than with basic initialisation. However, due to the large number of units produced by the transformation algorithm presented in Section 6.3.2, it is advisable not to use a too fine-grained approximation for the transformation algorithm, since otherwise the gain in initial accuracy is paid for by an excessive number of units that have to be removed in the early phase of training.
- *Inappropriate* initialisation can lengthen the training process, but due to its flexible structure, a Fine Blend can adapt without producing unnecessarily many units.

Algorithm 18: The $refine$ procedure for the Fine Blend

```
Uses: Fine Blend $N$ with units $\mathcal{U}$
      Input weight vectors $\boldsymbol{weightsIn}$
      Output weight vectors $\boldsymbol{weightsOut}$
      Error Information $Inf_{error}$
      $error_{j,d}$ values and $\boldsymbol{init}_{j,d}$ vectors (Initialisation Information)
      $utility(u)$ for any unit $u$ (Utility Information)

1  Determine unit $u \in \mathcal{U}$ with maximum error according to $Inf_{error}$
2  $\boldsymbol{pos} \leftarrow \boldsymbol{weightsIn}_u$
3  $P \leftarrow \left\{\boldsymbol{weightsIn}_v \middle| v \in \mathcal{U} \setminus \{u\}\right\}$
4  if $H_{ex}(\boldsymbol{pos}, P)$ exists then
5      $H \leftarrow H_{ex}(\boldsymbol{pos}, P)$
6  else
7      $H \leftarrow H_{in}(\{\boldsymbol{pos}\})$
8  end
9  $\boldsymbol{c} \leftarrow$ centre of $H$
10 Determine the (unique) hyper-square $H' \subseteq H$ with centre $\boldsymbol{c}'$
11    such that
12       level of $H' = 1 +$ level of $H$
13    and for all $j \in \{1, \ldots, m\}$
14       $c'_j < c_j$   iff   $error_{j,down}(u) > error_{j,up}(u)$
15 $\boldsymbol{out} \leftarrow \boldsymbol{0}$
16 foreach $j \in \{1, \ldots, m\}$ do
17     if $c'_j < c_j$ then
18         $\boldsymbol{out} \leftarrow \boldsymbol{out} + \boldsymbol{init}_{j,down}(u)$
19     else
20         $\boldsymbol{out} \leftarrow \boldsymbol{out} + \boldsymbol{init}_{j,up}(u)$
21     end
22 end
23 $\boldsymbol{weightsIn}_u \leftarrow \boldsymbol{c}$
24 Add a hidden unit $u'$ to $\mathcal{U}$ with
25    $\boldsymbol{weightsIn}_{u'} = \boldsymbol{c}'$
26    $\boldsymbol{weightsOut}_{u'} = \frac{1}{m} \cdot \boldsymbol{out}$
27 Reset $Inf_{error}$, $Inf_{init}$, $Inf_{util}$ for $u$ and $u'$
```

6.3.6 Wish Lists Reviewed

In this section, we will review the wish lists from Sections 6.1.6 and 6.2.6, explaining for each item how we have satisfied it.

Flexibility (6.1.6) The ***scaling*** parameters used by Cuboid Networks to determine the size of the hidden units' receptive areas are not necessary here, and the connection weights ***weightsIn*** from the input units to the hidden units, which encode the centres of the receptive areas and which were computed explicitly for Cuboid Networks, are learned by Fine Blend. The use of domain knowledge has been shifted into the $distance$ function of the hidden units' input function. We have thus gained the desired flexibility and enabled the hidden units to adapt their positions in the input space, while still exploiting our domain knowledge.

Robustness (6.1.6) The increased flexibility remedies two sources of fragility with Cuboid Networks, namely corruption of the ***scaling*** and of the ***weightsIn*** vectors. While the former have been abolished completely, the latter are now repaired automatically because they are obtained through training anyway. The awkward consequences of unit failure described in Section 6.1.5 are not an issue here: Due to the winner-takes-all behaviour, the mentioned holes in the input space cannot occur, and the failing units are automatically replaced through refinement or by moving existing units. Unnecessary refinements possibly caused by damage are detected and undone.

The increased flexibility and the unit removal mechanism thus make the Fine Blend less susceptible to damage and noise, and enable it to recover from their effects.

Unit Removal (6.1.6, 6.2.6) The utility controlled unit removal mechanism detects and removes unnecessary units which may have been used in earlier stages of training, wrongly generated as a consequence of damage to the network, or resulted from inappropriate initialisation. This also holds for the specific cases of unnecessary units described in 6.2.6: Through the global decay of all units' utilities over time, dead units will at some point be detected and removed; and since the utility increase for the winner unit depends on its contribution to the network output, the utility of idempotent units does not increase, so they are detected and removed as well. The requirement that one unit should be retained is taken into account by a slightly random initialisation of the utility values. Although the retained unit is not necessarily the central unit, the specific input function we use for the hidden units decreases the chance that an unsuitable unit from some other area takes over the area cleared of idempotent units.

Output Function (6.2.6) The Fine Blend uses the *winner-takes-all* behaviour, which is well-suited for computing partially constant functions and drastically reduces the number of hidden units needed in our specific application.

Error Computer (6.2.6) The general Error Computer utilising the tolerance vector $\boldsymbol{\epsilon}$ is used, allowing to discard negligibly small errors and thus helping to reach a stable network topology in always-on scenarios as described in Section 5.2.8.

Refinement Criterion (6.2.6) The *periodical threshold refinement* is used, avoiding unnecessarily precise refinements in favour of a stable network topology in always-on scenarios.

Internal Topology (6.2.6) The Fine Blend does not use an internal topology on the hidden units. In addition to the reduced administrative overhead as compared to SGNG, the caveat for robustness described in Section 6.2.5 is thus not an issue here.

7 Evaluation

7.1 Statistics

In this section, we will compare the systems described in Section 6 using statistics gathered from the implementation. All simulations use our running example program for rough initialisation and training, and an incorrect program for inappropriate initialisation and semantic noise. For convenience, the example program and the multi-dimensional level mapping are listed again in Figure 14, along with the incorrect program.

Correct program:

$$\begin{array}{l} e(0). \\ o(X) \leftarrow \neg e(X). \\ e(s(X)) \leftarrow o(X). \end{array}$$

Incorrect program:

$$\begin{array}{l} o(X) \leftarrow e(X). \\ e(s(X)) \leftarrow \neg o(X). \end{array}$$

Multi-dimensional level mapping:

$$\begin{array}{ll} \|e(s^n(0))\| := n+1 & \dim(e(s^n(0))) := 1 \\ \|o(s^n(0))\| := n+1 & \dim(o(s^n(0))) := 2 \end{array}$$

Figure 14: The correct program along with the incorrect program and the level mapping, as used for the statistics.

In Table 2, all parameters used in the simulations are listed. They were partially chosen through comparing simulations, but since there are many degrees of freedom, this could not be done exhaustively. Some parameters were chosen intuitively, or in the case of SGNG, inspired by [Fri98], without claiming optimality for our application. Therefore, all comparisons can be considered subjective, and tweaking the parameters might improve the results to some extent.

Note that we use base 4 for the embedding because the resulting values are represented more straightforwardly in the computer's binary floating point numbers than with the base 3 used for the examples so far.

The simulations always follow the same scheme: First the network is initialised with basic, rough, or inappropriate initialisation, the latter using the incorrect program.

Common Parameters	
Base used for embedding	4
Tolerance vector $\boldsymbol{\epsilon}$	$(0.0025, 0.0025)$
Greatest relevant output levels	
used for rough initialisation	$(3, 3)$
used for inappropriate initialisation	$(3, 3)$
resulting from $\boldsymbol{\epsilon}$	(4,4)
Greatest input levels	
used for reference set in error computation	$(6, 5)$
used for training example generation	$(6, 6)$
Maximum numeric noise	$\pm(0.00065, 0.00065)$
Semantic noise ratio	0.01
Cycle length for periodical threshold refinement	100
Parameters for Cuboid Networks	
Error reduction factor $errorRed$	0.3
Threshold for periodical threshold refinement	0.3333
$learningRate$	0.3
Parameters for SGNG	
Winner position adaptation rate μ_w	0.05
Neighbour position adaptation rate μ_n	0.0006
Output adaptation rate η	0.3
Maximum age for edges a_{max}	88
Proportionality factor for width parameter σ	0.7
Error reduction factor for all units $errorRed$	0.0005
Error reduction factor for parents α	0.5
Bias unit	enabled
Common Parameters for Fine Blend	
Maximum level l_{max}	12
Position adaptation rate μ	0.05
Output adaptation rate η	0.3
Error reduction factor $errorRed$	0.3
Threshold for periodical threshold refinement	0.3333
$utilityThreshold$	0.2
Parameters for Fine Blend 1	
Relative number k of examples to survive	7
Parameters for Fine Blend 2	
Relative number k of examples to survive	3

Table 2: Parameters used for the statistics.

Then it is trained with examples randomly generated from the correct program, i.e. pairs consisting of an embedded interpretation which randomly contains atoms up to the specified greatest input levels, and the embedded consequence. The training examples are modified either by no, numeric, or semantic noise, as defined in Section 5.2. In case of semantic noise, the wrong examples are generated from the incorrect program, using the same method as for correct examples.

After each presented example, the current number of units as well as the average of the component-wise maximum of the error relative to ϵ made by the network with respect to the reference set is logged. The reference set here consists of all examples from the correct program up to the specified greatest input level. In our case, the reference set is a strict subset of the set of potential training examples. The diagrams always show the error as dashed line in a logarithmic scale ranging from 0.01 to 100, along with the number of units as continuous line, both plotted against the number of examples presented.

7.1.1 Cuboid Networks

We will first compare the variants of Cuboid Networks in order to find out whether they are, in terms of performance, sufficiently similar to regard only one of them in the further comparisons.

Figure 15 compares the hierarchical refinement options of adding one child at a time (AddOneChild) and adding all children in one step (AddAllChildren). The property mentioned in Section 6.1.4 is confirmed: The former option takes more refinement steps, while the latter option adds more units. Apart from that, the performance is very similar. With noise as well as with other initialisation methods, the situation is the same. This holds also for the comparison of the two splitting refinement options, splitting either one dimension at a time (SplitOneDimension) or all dimensions in one step (SplitAllDimensions).

Figure 16 shows an exemplary comparison of SplitOneDimension and AddOneChild. Again, the performances are very similar. The differences are:

- The former stabilises faster than the latter. The reason is that the former needs less refinement steps than the latter to achieve the same input space granularity.
- The former produces less units than the latter. This is due to the fact that with hierarchical refinement the new units are added, while with splitting refinement the new units replace the old unit.

See Figure 17 for an illustration of the reasons.

Again, the other initialisation methods and noisy examples yield similar results. Therefore, we will only consider one variant of Cuboid Networks from now on. The remaining architectures are not hierarchical and add only one unit in each refinement step. In order to achieve fair comparisons, we will thus choose the Cuboid Network variant which has the same properties, i.e. SplitOneDimension.

Figure 18 compares the various initialisation methods using SplitOneDimension without noise. While the difference between rough and inappropriate initialisation is very clear in the beginning, they stabilise almost equally fast, and both perform better

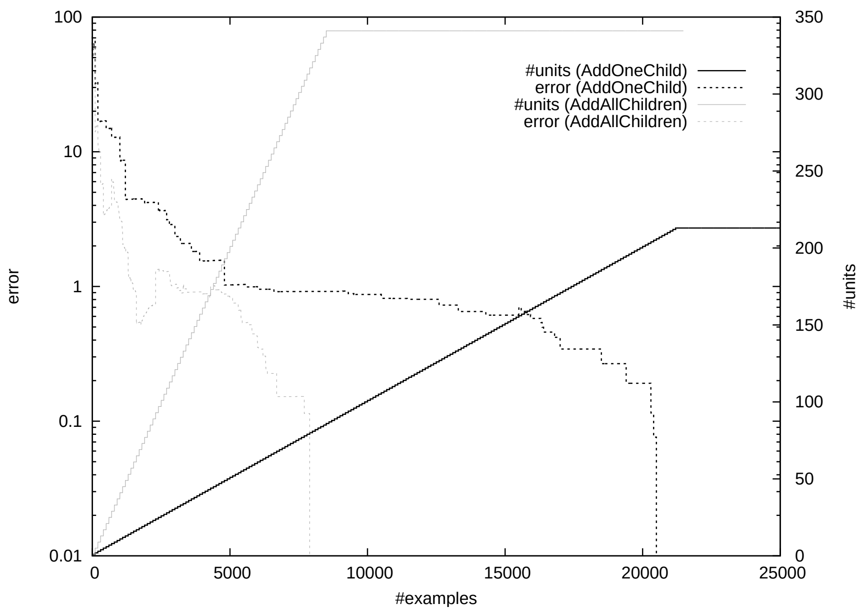


Figure 15: Comparison of AddOneChild and AddAllChildren, basic initialisation, no noise.

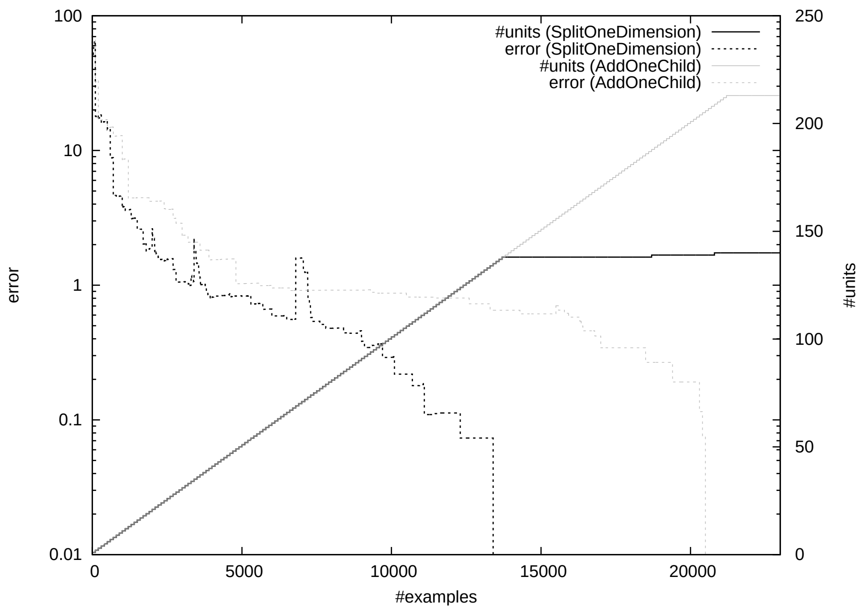


Figure 16: Comparison of SplitOneDimension and AddOneChild, basic initialisation, no noise.

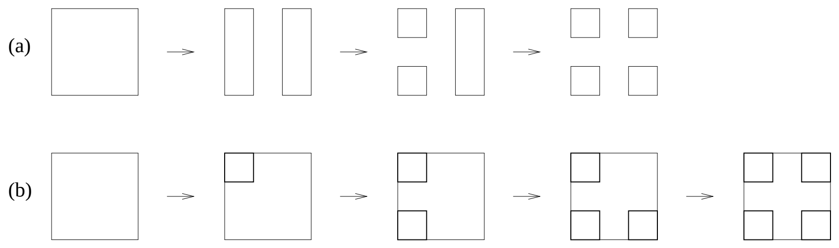


Figure 17: SplitOneDimension (a) needs 3 refinement steps to achieve the same input space granularity as AddOneChild (b) with 4 refinement steps. The former ends up with 4 units, the latter with 5.

than basic initialisation. This suggests that the critical issue is the generation of a sufficient number of units rather than their adjustment. Since the number of initially generated units does not suffice, all units have to be refined anyway, and thus their initially wrong output values with inappropriate initialisation do not matter.

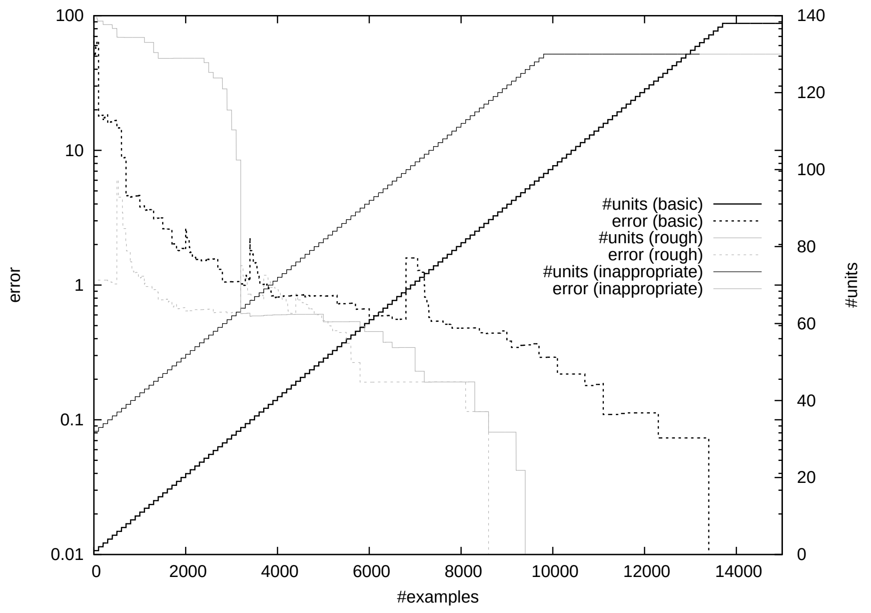


Figure 18: Comparison of the initialisation methods using SplitOneDimension, no noise. The numbers of units coincide for rough and inappropriate initialisation.

Figure 19 shows the same comparison with semantic noise. Initially, the different initialisation methods have an effect, but the network is able to compensate for the inappropriate initialisation. However, the number of units increases excessively, as anticipated in Section 6.1.5, and the noise disturbs the network enough to keep the error on a relatively high level.

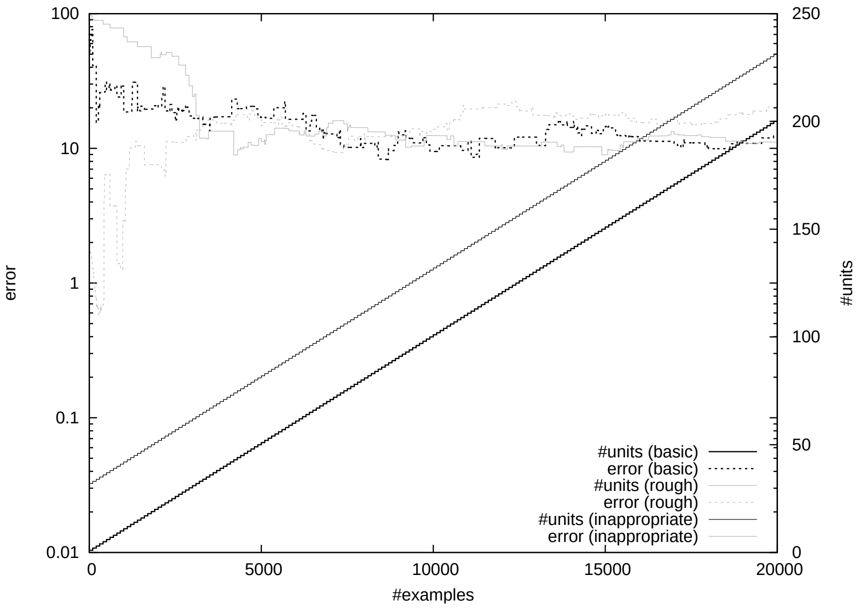


Figure 19: Comparison of the initialisation methods using SplitOneDimension, semantic noise. The numbers of units coincide for rough and inappropriate initialisation.

7.1.2 Fine Blend

For the Fine Blend, we use two setups: One with softer utility criteria (FineBlend 1) and one with stricter (FineBlend 2). The difference between these two setups becomes clear in Figure 20: Starting from rough initialisation, the former further decreases the error, paying with an increased number of units. The latter significantly decreases the number of units, paying with an increased error.

7.1.3 Fine Blend versus SGNG

Figure 21 compares FineBlend 1 with SGNG. Both start off similar, but soon SGNG fails to improve further. The excessively increasing number of units is partly due to the use of periodical refinement without threshold, but this should not be the cause for the constantly high error level. As mentioned above, the choice of parameters is rather subjective, and even though some testing was done to find them, they might be far from optimal. However, finding the optimal parameters for SGNG is beyond the scope of this thesis, and from the arguments in Section 6.2.6 it should be clear that it is not perfectly suited for our specific application. We will therefore concentrate on the remaining architectures in the following.

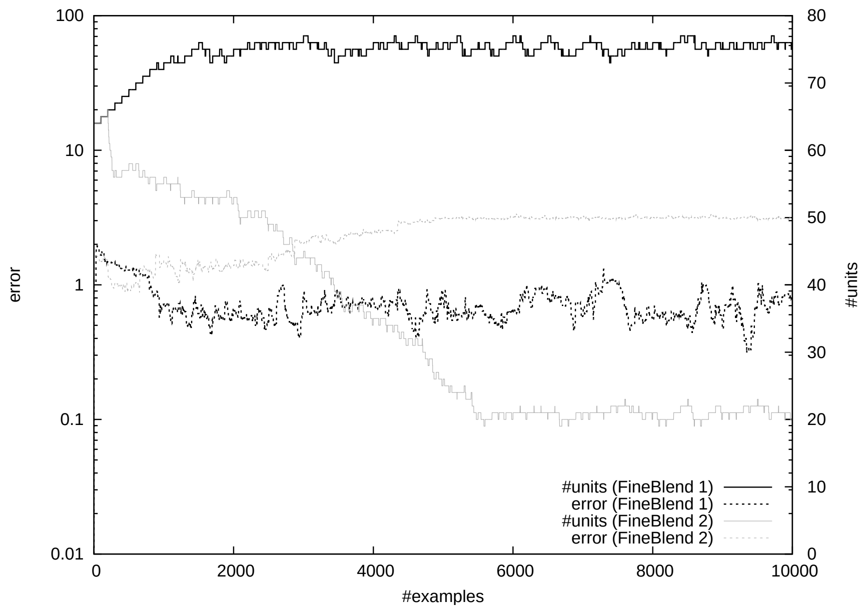


Figure 20: Comparison of FineBlend 1 and FineBlend 2, rough initialisation, numeric noise.

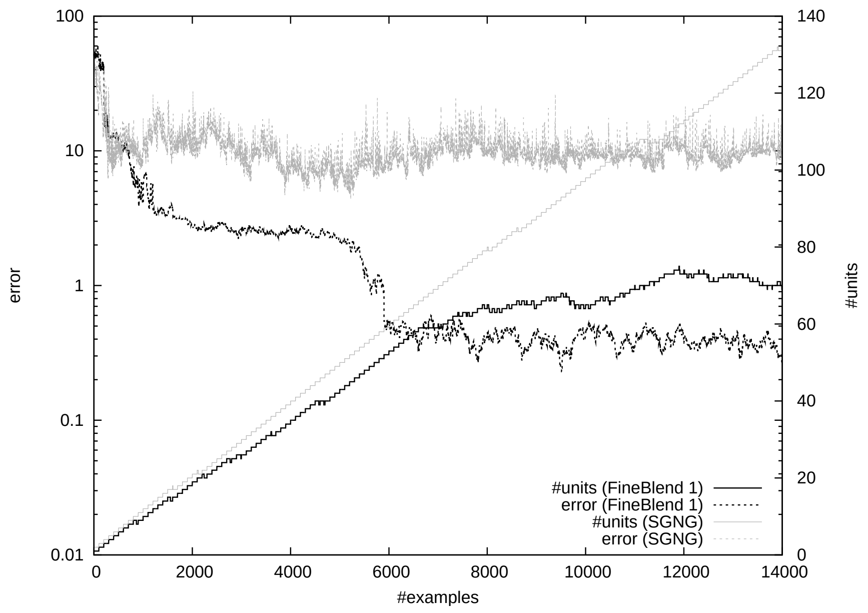


Figure 21: Comparison of FineBlend 1 and SGNG, basic initialisation, no noise.

7.1.4 Fine Blend versus Cuboid Networks

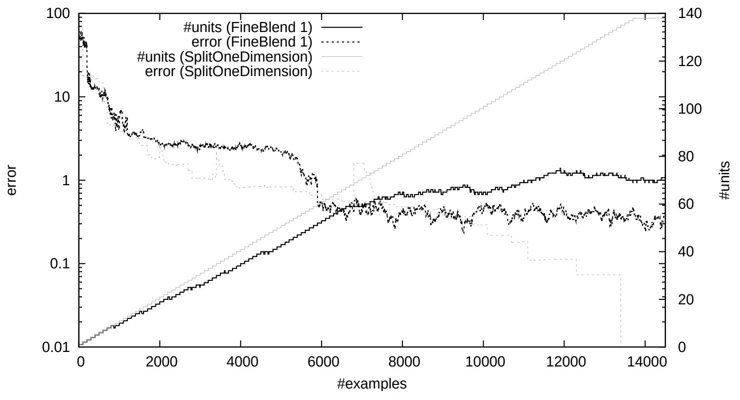
In this section, we will first compare the impact of the different initialisation methods in settings without noise and then examine the effects of noise in two exemplary experiments.

Figure 22 shows the effects of basic, rough, and inappropriate initialisation for FineBlend 1 and SplitOneDimension without noise. The following observations can be made:

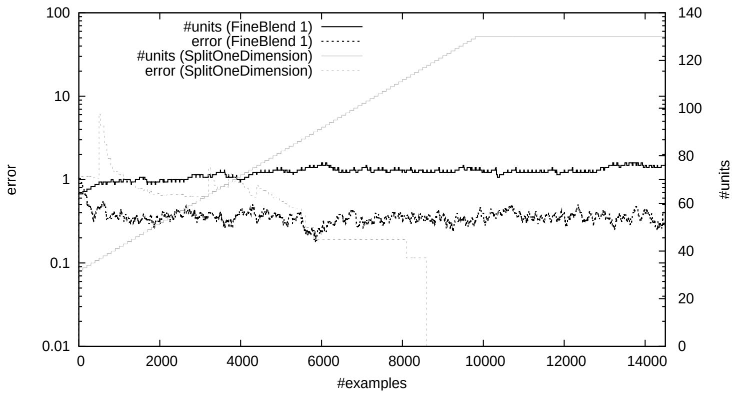
- The initialisation method does not affect the final outcome to a significant extent.
- With rough and with inappropriate initialisation, both networks stabilise significantly faster than with basic initialisation. This is mainly due to the fact that less units need to be generated.
- With rough initialisation, both networks stabilise only slightly faster than with inappropriate initialisation. This is due to the fact that the time needed to generating the necessary number of units, which is the same with both initialisations, outweighs the time needed to adjust the inappropriate initial output values.
- In the early phase of training, the difference between rough and inappropriate initialisation is clear: Both networks start with a significantly lower error. The early error peak in SplitOneDimension is probably due to premature refinements, i.e. there have not been sufficiently many examples to learn the correct output values for refining units.
- SplitOneDimension needs significantly longer than FineBlend 1 to correct the inappropriate initialisation. After the correction is done, the performance of both networks is very similar to the setting with rough initialisation.
- FineBlend 1 in all settings produces less units than SplitOneDimension, but also a higher relative error. However, the absolute error is still less than ϵ.

The effects of numeric noise are compared in Figure 23. FineBlend 1 is able to stabilise with a relatively low number of units and an error below 1. SplitOneDimension needs longer to reduce the error and is far less stable. At least it ceases to produce new units in the end, which is probably due to the fact that the noise values are lower than ϵ, so there is some state of the network where the noise only produces errors that are neglected.

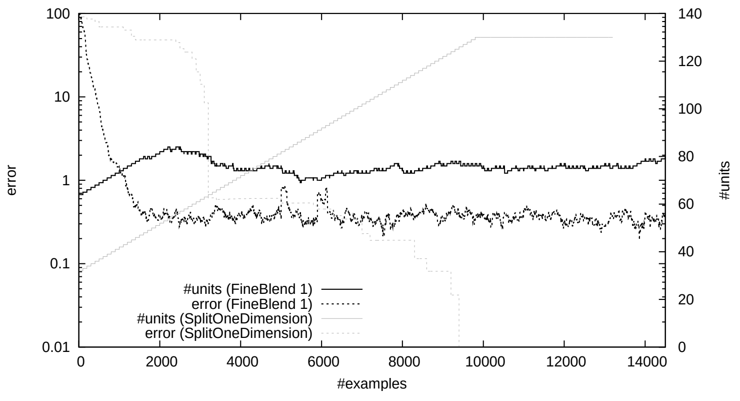
In our semantic noise setting, however, this is not the case, and thus the missing unit removal mechanism of SplitOneDimension results in a constantly increasing number of units. Figure 24 shows the results for this settings, comparing FineBlend 1, FineBlend 2, and SplitOneDimension. Obviously, the effect of completely wrong examples is much stronger than that of numerically slightly modified ones. Even with an ever-increasing number of units, SplitOneDimension has a constantly high error, while in both Fine Blend setups the struggle between insertion and removal of spurious units is clearly visible. Here, the stricter utility criteria of FineBlend 2 show to advantage: A constantly very small number of units is paid for with a rather slight increase in error (remember the error scale is logarithmic).



(a) basic initialisation



(b) rough initialisation



(c) inappropriate initialisation

Figure 22: Comparison of the initialisation methods using FineBlend 1 and SplitOneDimension, no noise.

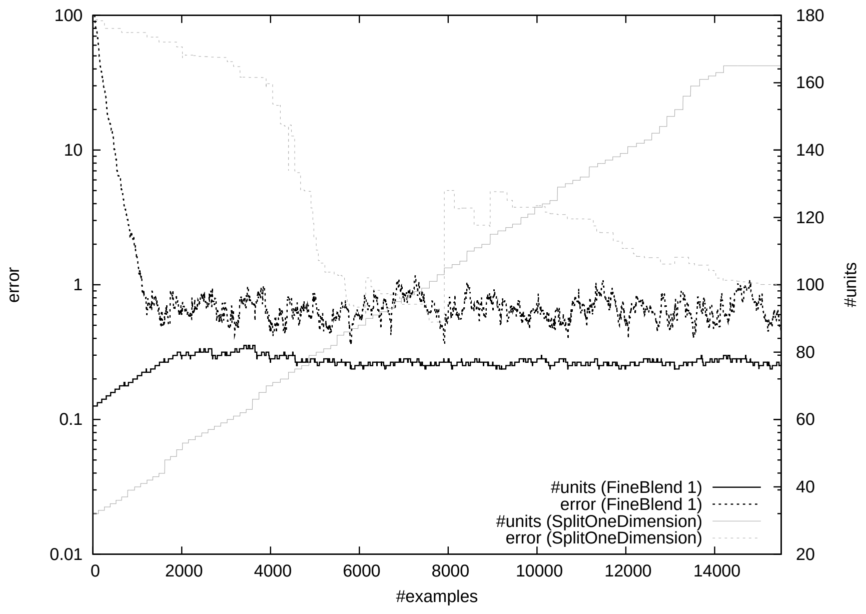


Figure 23: Comparison of FineBlend 1 and SplitOneDimension, inappropriate initialisation, numeric noise.

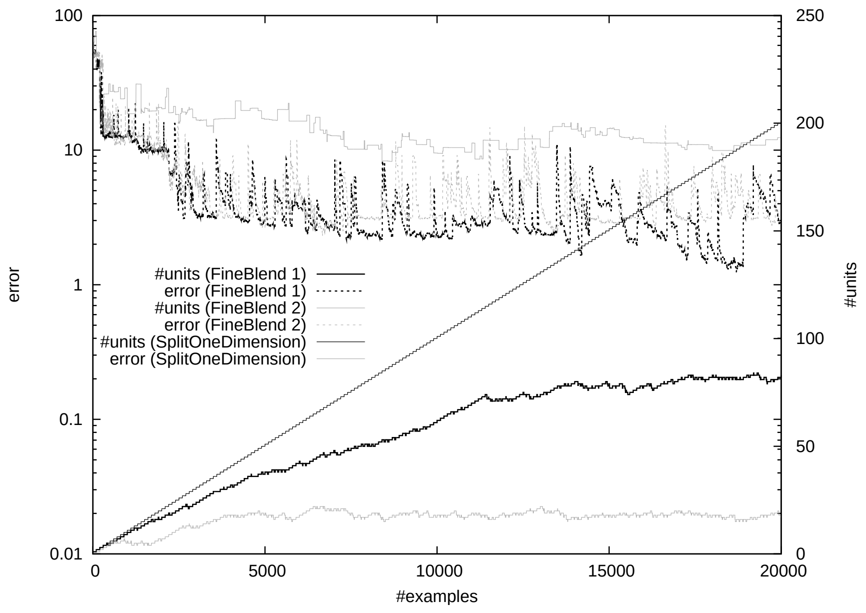


Figure 24: Comparison of FineBlend 1, FineBlend 2, and SplitOneDimension, basic initialisation, semantic noise.

7.1.5 Unit Failure

Figure 25 shows the effects of unit failure. This is simulated by randomly removing one third of the units of a fairly trained network, and then continuing training as if nothing had happened. The statistics are only presented for the Fine Blend, since the current implementations of the Cuboid Network training methods do not allow for unit removal. From Section 6.1.5, it should be clear that the Cuboid Networks are generally not able to recover from this kind of damage. In contrast, the Fine Blend proves to handle the damage gracefully.

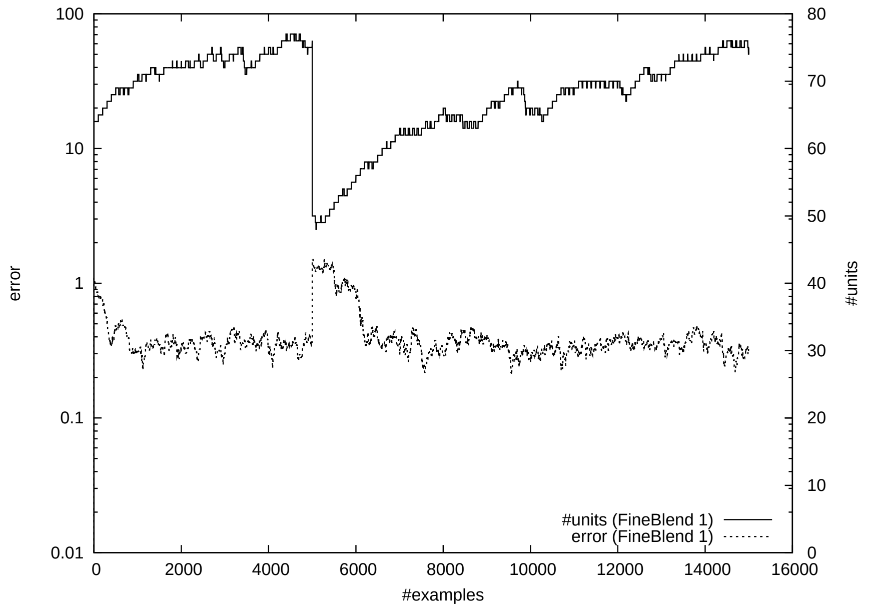


Figure 25: The effects of unit failure on FineBlend 1, rough initialisation, no noise. One third of the units were removed after 5000 examples.

All results shown so far confirm that the Fine Blend outperforms the two other architectures in our specific application and fulfils both wish lists, as stated in Section 6.3.6.

7.1.6 Iterating Random Inputs

One of our original aims was 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 the fixed point of the program's single-step operator, if the fixed point exists. In our running example, a unique fixed point is known to exist (see Example 3.4, Lemma 3.5, and the end of Section 2.1). Also, iterating the approximations we dealt with is guaranteed to approximate this fixed point, because our program is acyclic with respect to our multi-dimensional level mapping and we used the same greatest relevant output level for both dimensions (see Lemma 3.17). Since we proved the transformation algorithms to be correct, the interesting case is how networks trained from scratch behave.

We proceed as follows:

1. Train a network using the parameters given in Table 2 and the programs given in Figure 14, until the error caused by the network on the reference set described in Section 7.1 is 1. This means that, in average, network outputs are in the $\boldsymbol{\epsilon}$-neighbourhood of the desired output.

2. Transform the obtained network into a recurrent one by connecting the outputs to the inputs.

3. Choose a random input vector $\in D_{\mathbf{0}}$ (which is not necessarily a valid embedded interpretation) and use it as initial input to the network.

4. Iterate the network computation until the network reaches a stable state, i.e. until the outputs stay inside the $\boldsymbol{\epsilon}$-neighbourhood of the embedded fixed point of T_P.

For our example program, the unique fixed point of T_P is

$$\{e(0), o(s(0)), e(s^2(0)), o(s^3(0)), e(s^4(0)), \dots\},$$

(see Section 2.1), and its embedded value is

$$(0.10101010\ldots_4, 0.01010101\ldots_4) = (\tfrac{4}{15}, \tfrac{1}{15}) \approx (0.2666667, 0.0666667).$$

The $\boldsymbol{\epsilon}$-neighbourhood is thus

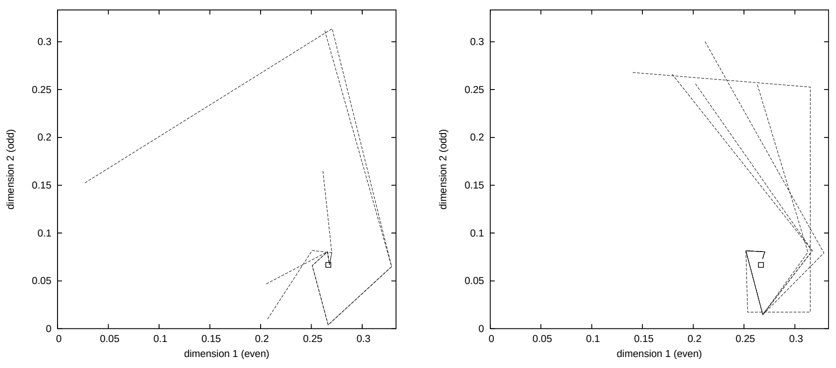
$$\left\{\boldsymbol{x} \in \mathbb{R}^m \middle| 0.2641667 < x_1 < 0.2691667 \quad \text{and} \quad 0.0641667 < x_2 < 0.0691667\right\}.$$

Figure 26 compares the results of this process for two FineBlend 1 networks, one trained without noise and one trained with semantic noise. The graphs represent the input space and show the $\boldsymbol{\epsilon}$-neighbourhood along with all intermediate results of the iteration for 5 random initial inputs in both cases.

Obviously the computation converges, and it does so fast. After at most 6 steps, the networks are stable in all cases, in fact they are completely stable in the sense that all outputs stay exactly the same and not only within a given neighbourhood. This corresponds roughly to the number of applications of our program's T_P required to fix the significant atoms, which confirms that the training methods really convey our intention of learning T_P.

The only difference between the network trained without noise and the network trained with semantic noise is that the latter is slightly mistaken with regard to the target vector, but still it settles to an equally stable state.

These results satisfy our requirements and expectations.



(a) trained without noise (b) trained with semantic noise

Figure 26: Iterating random inputs, FineBlend 1, basic initialisation. The two dimensions of the input vectors are plotted against each other. The ϵ-neighbourhood of the embedded fixed point is shown as a small box.

7.2 Properties in Comparison to ILP

While it would be presumptuous to compare the results we achieved in an area which is only just emerging from its infancy to the comparatively established techniques of Inductive Logic Programming (ILP), there are some distinguishing fundamental properties that we can describe. For the ILP side, these properties can be found in [NCdW97] unless cited otherwise.

The most prominent difference lies in the objective of learning. ILP systems are usually presented with positive and negative example atoms from some model and try to find a program of which only the positive examples are a logical consequence. Depending on the concrete semantics used, this could be regarded as learning a program via some fixed point of its single-step operator. In our approach of neural-symbolic integration, also called the Core method, we try to learn only one application of the single-step operator. Therefore, both the learning target and the kind of training examples differ fundamentally.

If the necessary kind of examples is available, then the Core method has some advantages:

- (Mutually) recursive predicates, which are a source of difficulties for a certain class of ILP systems as pointed out in [Bra99], do not cause any problems at all. Since we learn only one application of the single-step operator, recursive predicates can be handled exactly like any other ones.

- Consider the following two simple programs:

$$\boxed{\begin{array}{l} p \leftarrow q. \\ q. \end{array}} \quad \text{and} \quad \boxed{\begin{array}{l} p. \\ q. \end{array}}$$

Both have the same logical consequences p and q, but with the Core method, we can capture the causalities.

- For the program

$$p \leftarrow \neg q.$$
$$q \leftarrow \neg p.$$

different semantics exist, and the logical consequences depend on the chosen semantics. The Core method is independent of the chosen semantics.

Additionally, there are advantageous properties inherent to the connectionist learning approach:

- It is designed for incremental learning without the need of storing the examples encountered.
- Noise can be coped with to a certain degree, as shown in Section 7.

Both issues cause problems for many ILP systems.

Of course there is still the task to extract symbolic information, e.g. a logic program, from the systems we construct and train. Some of the advantages and desirable properties described so far may be lost in this process. But at least they could be established up to this stage of the neural-symbolic learning cycle and are not lost a priori.

8 Conclusion and Future Work

In this thesis, we first extended the results from [Wit05, BHW05] by lifting the real-valued approximations of covered first-order logic programs to multi-dimensional real vectors in order to facilitate an implementation on a computer with limited floating point precision (Section 3). We then tried to construct layered sigmoidal feed-forward networks approximating the functions obtained previously and came to the conclusion that this approach is not suitable (Section 4). Consequently, we decided to concentrate on locally receptive architectures and prepared an implementation, introducing a common network structure and a general training framework (Section 5). We then designed a specialised network architecture, reviewed an established general system, and used ideas from both in order to devise a second specialised architecture; we discussed the properties of all approaches and described implementations along with the appropriate transformation and training algorithms (Section 6). Finally, we evaluated the systems using statistics obtained from the implementations and put our approach in the context of Inductive Logic Programming (Section 7).

The evaluation confirmed the hypotheses formulated during the design process and showed that the last architecture outperforms the two previous ones and satisfies our expectations.

This concludes a prototypical realisation of the representation and training components of the neural-symbolic learning cycle [BH05] for covered logic programs. Besides implementing the remaining components of the cycle, it is also necessary to further examine the systems presented in this thesis.

The convergence behaviour when iterating values through the network, covered briefly in Section 7.1.6, should be studied more thoroughly. Furthermore, the tests were performed using very simple and low-dimensional examples, so it would be desirable to examine more involved cases closer to the real world. In order to be able to handle them, more efforts are required to find the right parameters, and some improvements to the architectures should be considered.

Interesting ideas for extensions include mechanisms to automatically adjust the learning parameters in the course of the training, or to explicitly limit the maximum number of units available. To facilitate the extraction of symbolic knowledge from the connectionist systems, it might be an advantage if common consequences could be captured even if not all consequences coincide, as described at the end of Section 6.1.4.

Finally, to obtain a frame of reference for judging the performance of these specialised systems, a comparison to a standard connectionist system trained with back-propagation would be desirable.

References

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian 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.

[BGH05] Sebastian Bader, Artur Garcez, and Pascal Hitzler. Computing first-order logic programs by fibring artificial neural networks. In I. Russell and Z. Markov, editors, *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Symposium Conference, Clearwater Beach, Florida, USA*, pages 314–319. AAAI Press, MAY 2005.

[BH04] Sebastian Bader and Pascal Hitzler. Logic programs, iterated function systems, and recurrent radial basis function networks. *Journal of Applied Logic*, 2(3):273–300, 2004.

[BH05] Sebastian Bader and Pascal Hitzler. Dimensions of neural-symbolic integration - a structured survey. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 1, pages 167–194. International Federation for Computational Logic, College Publications, JUL 2005.

[BHH04] Sebastian Bader, Pascal Hitzler, and Steffen Hölldobler. The integration of connectionism and knowledge representation and reasoning as a challenge for artificial intelligence. In L. Li and K.K. Yen, editors, *Proceedings of the Third International Conference on Information, Tokyo, Japan*, pages 22–33. International Information Institute, 2004. ISBN 4-901329-02-2.

[BHW05] Sebastian Bader, Pascal Hitzler, and Andreas Witzel. Integrating first order logic programs and connectionist systems - a constructive approach. In *In:*

Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, AUG 2005.

[Bra99] Ivan Bratko. Refining complete hypotheses in ILP. In *ILP '99: Proceedings of the 9th International Workshop on Inductive Logic Programming*, pages 44–55, London, UK, 1999. Springer-Verlag.

[Fri98] Bernd Fritzke. *Vektorbasierte Neuronale Netze*. Habilitation, Technische Universität Dresden, 1998.

[GBG02] Artur S. d'Avila Garcez, Krysia B. Broda, and Dov M. Gabbay. *Neural-Symbolic Learning Systems — Foundations and Applications*. Perspectives in Neural Computing. Springer, Berlin, 2002.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming. Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GZ99] Artur S. d'Avila Garcez and Gerson Zaverucha. The connectionist inductive lerarning and logic programming system. *Applied Intelligence, Special Issue on Neural networks and Structured Knowledge*, 11(1):59–77, 1999.

[HHS04] Pascal Hitzler, Steffen Hölldobler, and Anthony K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.

[HK94] Steffen Hölldobler and Yvonne Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.

[HKS99] Steffen Hölldobler, Yvonne Kalinke, and Hans-Peter Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999.

[HS03] Pascal Hitzler and Anthony K. Seda. Generalized metrics and uniquely determined logic programs. *Theoretical Computer Science*, 305(1–3):187–219, 2003.

[Llo88] John W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1988.

[NCdW97] S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer, 1997.

[Roj96] R. Rojas. *Neural Networks — A Systematic Introduction*. Springer, 1996.

[Wit05] Andreas Witzel. Integrating first-order logic programs and connectionist systems - a constructive approach. Project thesis, Technische Universität Dresden, 2005.