Integrating First-Order Logic Programs and Connectionist Systems — A Constructive Approach

Sebastian Bader[1*], **Pascal Hitzler**[2†], **Andreas Witzel**[3]
[1]International Center for Computational Logic, Technische Universität Dresden, Germany
[2]AIFB, Universität Karlsruhe, Germany
[3]Department of Computer Science, Technische Universität Dresden, Germany

Abstract

Significant advances have recently been made concerning the integration of symbolic knowledge representation with artificial neural networks (also called connectionist systems). However, while the integration with propositional paradigms has resulted in applicable systems, the case of first-order knowledge representation has so far hardly proceeded beyond theoretical studies which prove the existence of connectionist systems for approximating first-order logic programs up to any chosen precision. Advances were hindered severely by the lack of concrete algorithms for obtaining the approximating networks which were known to exist: the corresponding proofs are not constructive in that they do not yield concrete methods for building the systems. In this paper, we will make the required advance and show how to obtain the structure and the parameters for different kinds of connectionist systems approximating covered logic programs.

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.

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), as extraction methods should easily follow from successful methods for encoding.

For propositional logic programs, encodings into connectionist systems like [11] led immediately to applicable algorithms. Corresponding learning paradigms have been developed [7; 6] and applied to real settings.

For the first-order logic case, however, the situation is much more difficult, as laid out in [4]. Concrete translations, as in [3; 2], yield nonstandard network architectures. For standard architectures, previous work has only established non-constructive proofs showing the existence of connectionist systems which approximate given logic program with arbitrary precision [12; 9]. Thus the implementation of first-order integrated systems was impossible up to this point.

In this paper, we will give concrete methods to compute the structure and the parameters of connectionist systems approximating certain logic programs using established standard architectures.

First, in Section 2, we will give a short introduction to logic programs and connectionist systems. We also review 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 [12]. In Section 3, we will then approximate the resulting real function by

*Sebastian Bader is supported by the GK334 of the German Research Foundation (DFG).

†Pascal Hitzler is supported by the German Federal Ministry of Education and Research (BMBF) under the SmartWeb project, and by the European Union under the KnowledgeWeb Network of Excellence.

a piecewise constant function in a controlled manner, which is an important simplifying step for establishing our results. We will then construct connectionist systems for computing or approximating this function, using sigmoidal activation functions in Section 4 and radial basis function (RBF) architecture in Section 5. Section 6 will conclude the paper with a short discussion of some open problems and possibilities for future work.

2 Preliminaries

In this section, we shortly review the basic notions needed from logic programming and connectionist systems. Main references for background reading are [13] and [14], respectively. We also review the embedding of T_P into the real numbers as used in [12; 9], and on which our approach is based.

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*
$e(s(X)) \leftarrow \neg e(X)$	*% the successor $s(X)$*
	% of a non-even 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. 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 $\|\cdot\| : \mathcal{B}_P \to \mathbb{N} \setminus \{0\}$. In this paper, we require level mappings to be injective, in which case they can be thought of as enumerations of $\mathcal{B}_P$. 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. All acyclic programs are also covered under our standing condition that level mappings are injective, and provided that function symbols are present, i.e. $\mathcal{B}_P$ is 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 [11; 7].

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

$$\begin{aligned}\mathcal{U}_P &= \{0, s(0), s^2(0), \dots\}\\ \mathcal{B}_P &= \{e(0), e(s(0)), e(s^2(0)), \dots\}\\ \mathcal{G}(P) &= e(0).\\ &\quad\ e(s(0)) \leftarrow \neg e(0).\\ &\quad\ e(s^2(0)) \leftarrow \neg e(s(0)).\\ &\quad\ \vdots\end{aligned}$$

With $\|e(s^n(0))\| := n + 1$, 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:*

$$\begin{aligned}I_1 &= \{e(0), e(s(0))\}\\ I_2 &= \{e(0), e(s^3(0)), e(s^4(0)), e(s^5(0)), \dots\}\\ I_3 &= \{e(0), e(s^2(0)), e(s^4(0)), e(s^6(0)), \dots\}\\ I_4 &= \mathcal{B}_P\end{aligned}$$

$I_1 \not\models P$ since $e(s^3(0)) \leftarrow \neg e(s^2(0)) \in \mathcal{G}(P)$ and $e(s^2(0)) \notin I_1$, but $e(s^3(0)) \notin I_1$. I_2 is neither a model (for a similar reason). Both I_3 and I_4 are models for 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 [1]. The well-known stable models [8], for example, are always supported. In example 2.1, $I_3 = \{e(0), e(s^2(0)), e(s^4(0)), \dots\}$ is supported (and stable), while $I_4 = B_P$ is a model but not supported.

Example 2.4. *For P from Example 2.1 and I_1, I_2 from Example 2.3, we get the following by successive application (i.e. iteration) of T_P:*

$$\begin{aligned}I_1 \overset{T_P}{\mapsto} I_2 \overset{T_P}{\mapsto} \{e(0), e(s^2(0)), e(s^3(0))\} \overset{T_P}{\mapsto} \dots\\ \overset{T_P}{\mapsto} \{e(0), e(s^2(0)), \dots, e(s^{2n}(0)), e(s^{2n+1}(0))\} \overset{T_P}{\mapsto} \dots\end{aligned}$$

For a certain class of programs, the process of iterating T_P can be shown to converge[1] to the unique supported Herbrand

[1] 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 [10], where also a general class of programs, called Φ-*accessible programs*, is described, for which iterating T_P always converges in this sense.

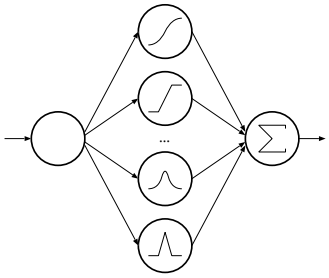

Figure 1: A simple 3-layered feed-forward connectionist system, with different activation functions depicted in the hidden layer.

model of the program, which in this case is the model describing the semantics of the program [10]. 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[2] 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 iterates converge in this sense to $I_3 = \{e(0), e(s^2(0)), e(s^4(0)), \dots\}$, 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*. We will deal with feed-forward networks, i.e. networks without cycles, as shown in Figure 1. Each unit has an *input function* which merges its inputs into one input using the weights, and an *activation function* which then computes the output. If a unit has inputs $x_1, \dots, x_n$ with weights $w_1, \dots, w_n$, then the *weighted sum* input function is $\sum_{i=1}^{n} x_i w_i$. A locally receptive *distance* input function is $\sqrt{\sum_{i=1}^{n}(x_i - w_i)^2}$. In the case of one single input, this is equivalent to $|x_1 - w_1|$. Those units without incoming connections are called *input neurons*, those without outgoing ones are called *output neurons*.

2.3 Embedding T_P in $\mathbb{R}$

As connectionist systems propagate real numbers, and single-step operators map interpretations, i.e. subsets of $\mathcal{B}_P$, we need to bridge the gap between the real-valued and the symbolic setting. We follow the idea laid out first in [12], and further developed in [9], for embedding $\mathcal{I}_P$ into $\mathbb{R}$. For this purpose, we define $R : \mathcal{I}_P \to \mathbb{R}$ as $R(I) := \sum_{A \in I} b^{-\|A\|}$ for some base $b \geq 3$. Note that R is injective. We will abbreviate $R(\{A\})$ by $R(A)$ for singleton interpretations. As depicted in Figure 2, we obtain f_P as an *embedding of* T_P *in* $\mathbb{R}$: $f_P : D_f \to D_f$ with $D_f := \{R(I) | I \in \mathcal{I}_P\}$, is defined as $f_P(x) := R(T_P(R^{-1}(x)))$. Figure 3 shows the graph of the

[2] In this case the level mapping does not need to be injective.

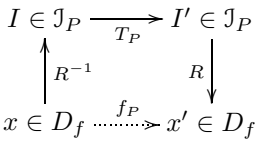

Figure 2: Relations between T_P and f_P

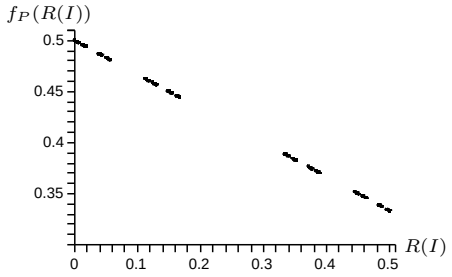

Figure 3: The graph of the embedded T_P-operator from Example 2.1, using base 3 for the embedding. In general, the points will not be on a straight line.

embedded T_P-operator associated to the program discussed in Examples 2.1 to 2.4.

3 Constructing Piecewise Constant Functions

In the following, we assume P to be a covered program with bijective level mapping $\|\cdot\|$ which is, along with its inverse $\|\cdot\|^{-1}$, effectively computable. As already mentioned, we also assume that $\mathcal{B}_P$ is infinite. However, our approach will also work for the finite case with minor modifications. Furthermore, R and f_P denote embeddings with base b as defined above.

3.1 Approximating one Application of T_P

Within this section we will show how to construct a ground subprogram approximating a given program. I.e., we will construct a subset P_l of the ground program $\mathcal{G}(P)$, such that the associated consequence operator T_{P_l} approximates T_P up to a given accuracy ϵ. This idea was first proposed in [15].

Definition 3.1. For all $l \in \mathbb{N}$, the set of atoms of level less than or equal to l is defined as $\mathcal{A}_l := \{A \in \mathcal{B}_P | \|A\| \leq l\}$. Furthermore, we define the *instance of P up to level l* as $P_l := \left\{A \leftarrow \text{body} \in \mathcal{G}(P) \middle| A \in \mathcal{A}_l\right\}$.

Since the level mappings are required to be enumerations, we know that $\mathcal{A}_l$ is finite. Furthermore, it is also effectively computable, due to the required computability of $\|\cdot\|^{-1}$. It is clear from the definition that P_l is ground and finite, and again, can be computed effectively.

Definition 3.2. For all $l \in \mathbb{N}$, the *greatest relevant input level* with respect to l is

$$\hat{l} := \max\left\{\|L\| \middle| L \text{ is a body literal of some clause in } P_l\right\}.$$

Obviously, we can compute $\hat{l}$ easily, since P_l is ground and finite. The following lemma establishes a connection between the consequence operators of some ground subprogram P_k and the original program P.

Lemma 3.3. For all $l, k \in \mathbb{N}$, $k \geq l$, and $I, J \in \mathfrak{I}_P$, we have that $T_{P_k}(I)$ and $T_P(J)$ agree on $\mathcal{A}_l$ if I and J agree on $\mathcal{A}_{\hat{l}}$, i.e.

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

Proof. This follows simply from the fact that I and J agree on $\mathcal{A}_{\hat{l}}$, and that T_{P_k} contains all those clauses relating atoms from $\mathcal{A}_{\hat{l}}$ and $\mathcal{A}_l$. Taking this into account we find that T_P and T_{P_k} agree on $\mathcal{A}_l$. □

Definition 3.4. The *greatest relevant output level* with respect to some arbitrary $\epsilon > 0$ is

$$\begin{aligned} o_\epsilon &:= \min \left\{ n \in \mathbb{N} \middle| \sum_{\|A\|>n} R(A) < \epsilon \right\} \\ &= min \left\{ n \in \mathbb{N} \middle| n > -\frac{\ln(b-1)\epsilon}{\ln b} \right\} \end{aligned}$$

The following theorem connects the embedded consequence operator of some subprogram with a desired error bound, which will be used for later approximations using neural networks.

Theorem 3.5. For all $\epsilon > 0$, we have that

$$\left| f_P(x) - f_{P_{o_\epsilon}}(x) \right| < \epsilon \quad \text{for all } x \in D_f.$$

Proof. Let $x \in D_f$ be given. From Lemma 3.3, we know that $T_{P_{o_\epsilon}}(R^{-1}(x)) = R^{-1}(f_{P_{o_\epsilon}}(x))$ agrees with $T_P(R^{-1}(x)) = R^{-1}(f_P(x))$ on all atoms of level $\leq o_\epsilon$. Thus, $f_{P_{o_\epsilon}}(x)$ and $f_P(x)$ agree on the first o_ϵ digits. So the maximum deviation occurs if all later digits are 0 in one case and 1 in the other. In that case, the difference is $\sum_{\|A\|>n} R(A)$, which is $< \epsilon$ by definition of o_ϵ. □

3.2 Iterating the Approximation

Now we know that one application of $f_{P_{o_\epsilon}}$ approximates f_P up to ϵ. But what will happen if we try to approximate several iterations of f_P? In general, $\hat{o}_\epsilon$ might be greater than o_ϵ, that is, the required input precision might be greater than the resulting output precision. In that case, we 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 3.6. For all $l, n \in \mathbb{N}$, we can effectively compute $l^{(n)}$ such that for all $I \in \mathfrak{I}_P$, $m \leq n$, and $k \geq l^{(n)}$:

$$T^m_{P_k}(I) \text{ agrees with } T^m_P(I) \text{ on } \mathcal{A}_l.$$

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

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

step $n \rightsquigarrow n+1$**:** By induction hypothesis, we can find $l^{(n)}$ such that for all $I \in \mathfrak{I}_P$, $m \leq n$, and $k \geq l^{(n)}$, $T^m_{P_k}(I)$ agrees with $T^m_P(I)$ on $A_{\hat{l}}$. With $l^{(n+1)} := \max\{l, l^{(n)}\}$, we then have for all $I \in \mathfrak{I}_P$, $m \leq n$, and $k \geq l^{(n+1)}$:

$$\begin{aligned} & T^m_{P_k}(I) \text{ agrees with } T^m_P(I) \text{ on } A_{\hat{l}} && (k \geq l^{(n)}) \\ \Rightarrow \quad & T^{m+1}_{P_k}(I) \text{ agrees with } T^{m+1}_P(I) \text{ on } \mathcal{A}_l && (3.3) \end{aligned}$$

$T^0_{P_k}(I) = I = T^0_P(I)$ completes the Induction Step. □

$$x = 0.\underbrace{0010101101010010}_{\hat{l} \text{ digits are equal}}000000\ldots_b$$

$$x' = 0.\overbrace{0010101101010010}111111\ldots_b$$

Figure 4: Example for the endpoints of a range $[x, x']$ on which f_{P_l} is constant

It follows that for all $\epsilon > 0$, we can effectively compute $o^{(n)}_\epsilon$ such that $\left| f^n_P(x) - f^n_{P_{o^{(n)}_\epsilon}}(x) \right| < \epsilon$ for all $x \in D_f$.

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 so many iterations anyway, since T_P does not necessarily converge.

For acyclic programs, however, T_P is guaranteed to converge, and additionally we can prove that we do not lose precision in the application of T_{P_l}. Due to the acyclicity of P we have $\hat{l} < l$, and hence, with respect to $\mathcal{A}_l$, we obtain the same result after n iterations of T_{P_l} as we would obtain after n iterations of T_P. Thus we can approximate the fixed point of T_P by iterating T_{P_l}. To put it formally, we have that $T^n_{P_l}(I)$ agrees with $T^n_P(I)$ on $\mathcal{A}_l$ for acyclic P and all $n \in \mathbb{N}$. Thus, in this case we find that $|f^n_P(x) - f^n_{P_{o_\epsilon}}(x)| < \epsilon$ for all $x \in D_f$ and all $n \in \mathbb{N}$.

3.3 Simplifying the Domain

Now we have gathered all information and methods necessary to approximate f_P and iterations of f_P. 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_f to some larger set D_l.

The idea is as follows. Since only input atoms of level $\leq \hat{l}$ play a role in P_l, we have that all $x \in D_f$ which differ only after the $\hat{l}$-th digit are mapped to the same value by f_{P_l}. So we have ranges $[x, x'] \subseteq \mathbb{R}$ of fixed length with x and x' as in Figure 4 such that all elements of $[x, x'] \cap D_f$ are mapped to the same value. Obviously, there are $2^{\hat{l}}$ such ranges, each of length $\sum_{\|A\|>\hat{l}} R(A)$. So we can extend f_{P_l} to a function $\hat{f}_{P_l}$ which has a domain consisting of $2^{\hat{l}}$ disjoint and connected ranges and is constant on each of these ranges. Additionally, the minimum distance between two ranges is greater than or equal to the length of the ranges.

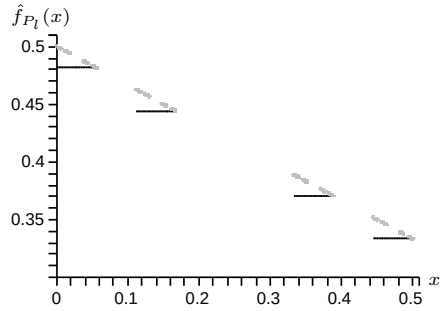

Figure 5: Example for the graph of $\hat{f}_{P_l}$ with $\hat{l} = 2$; f_P is shown in grey.

The resulting graph of $\hat{f}_{P_l}$ will then look similar to the one shown in Figure 5. We formalise these results in the following.

Definition 3.7. An ordered enumeration of all left borders $d_{l,i}$ can be computed as

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

Each of the intervals has length

$$\lambda_l := \sum_{\|A\|>\hat{l}} R(A) = \frac{1}{(b-1)\cdot b^{\hat{l}}}.$$

Finally, we define

$$D_l := \bigcup_{i=0}^{2^{\hat{l}}-1} D_{l,i} \qquad \text{with } D_{l,i} := [d_{l,i}, d_{l,i} + \lambda_l].$$

Thus, D_l consists of $2^{\hat{l}}$ pieces of equal length.

Lemma 3.8. For all $l \in \mathbb{N}$, we have $D_l \supseteq D_f$.

Proof. Let $l \in \mathbb{N}$ and $x \in D_f$. Then there is a $d_{l,i}$ which agrees with x on its $\hat{l}$ digits. But $D_{l,i}$ contains all numbers which agree with $d_{l,i}$ on its $\hat{l}$ digits, thus $x \in D_{l,i} \subseteq D_l$. □

Lemma 3.9. For all $l \in \mathbb{N}$, the connected parts of D_l do not overlap 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. $\hat{l}$-th, digit. In that case, the distance between these endpoints is $b^{-\hat{l}}$, which is $\geq 2 \cdot \lambda_l$ since $b \geq 3$. □

Lemma 3.10. For all $l \in \mathbb{N}$ and $0 \leq i < 2^{\hat{l}}$, f_{P_l} is constant on $D_{l,i} \cap D_f$.

Proof. All atoms in bodies of clauses in P_l are of level $\leq \hat{l}$. Thus, T_{P_l} regards only those atoms of level $\leq \hat{l}$, i.e. T_{P_l} is constant for all interpretations which agree on these atoms. This means that f_{P_l} is constant for all x that agree on the first $\hat{l}$ digits, which holds for all $x \in D_{l,i} \cap D_f$. □

Definition 3.11. The *extension of* f_{P_l} *to* D_l , $\hat{f}_{P_l} : D_l \to D_f$, is defined as $\hat{f}_{P_l}(x) := f_{P_l}(d_{l,i})$ for $x \in D_{l,i}$. From the results above, it follows that $\hat{f}_{P_l}$ is well-defined.

Now we have simplified the domain of the approximated embedded single-step operator such that we can regard it as a function consisting of a finite number of equally long constant pieces with gaps at least as wide as their length.

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

4 Constructing Sigmoidal Feed-Forward Networks

We will continue our exhibition by considering some arbitrary piecewise constant function g which we want to approximate by connectionist systems. Since $\hat{f}_{P_l}$ is piecewise constant, we can treat this function as desired, and others by the same method. So in the following, let $g : D \to \mathbb{R}$ be given by

$$D := \bigcup_{i=0}^{n-1} [a_i, c_i], \quad c_i = a_i + b, \quad c_i < a_{i+1},$$
$$g(x) := y_i \text{ for } x \in [a_i, c_i].$$

When we construct our connectionist systems, we are only interested in the values they yield for inputs in D. We do not care about the values for inputs outside of D since such inputs are guaranteed not to be possible embeddings of interpretations, i.e. in our setting they do not carry any symbolic meaning which can be carried back to $\mathfrak{I}_P$.

We will proceed in two steps. First, we will approximate g by using connectionist systems with step activation functions. Afterwards, we will relax our approach for the treatment of sigmoidal activation functions.

4.1 Step Activation Functions

We will now construct a multi-layer feed-forward network with weighted sum input function, where each of the units in the hidden layer computes the following step function:

$$s_{l,h,m}(x) := \begin{cases} l & \text{if } x \leq m \\ l+h & \text{otherwise.} \end{cases}$$

As an abbreviation, we will use $s_i(x) := s_{l_i,h_i,m_i}(x)$ for $0 \leq i < n-1$. We want the output to agree with g on its domain, that is, we want $\sum_{i=0}^{n-2} s_i(x) = g(x)$ for all $x \in D$.

An intuitive construction is depicted in Figure 6. For n pieces, we use $n-1$ steps. We put one step in the middle between each two neighbouring pieces, then obviously the height of that step must be the height difference between these two pieces.

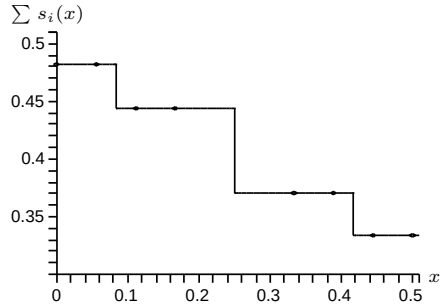

Figure 6: Sum of the step functions.

It remains to specify values for the left arms of the step functions. All left arms should add up to the height of the first piece. So we can choose that height divided by $n-1$ for each left arm. Now we have specified all s_i completely:

Definition 4.1. For $0 \leq i < n-1$,

$$l_i := \frac{y_0}{n-1}; \quad h_i := -y_i + y_{i+1}; \quad m_i := \frac{1}{2}(c_i + a_{i+1})$$

Theorem 4.2. $\sum_{i=0}^{n-2} s_i(x) = g(x)$ for all $x \in D$.

Proof. Let $x \in [a_j, c_j]$. Then

$$\sum_{i=0}^{n-2} s_i(x) = \sum_{i=0}^{j-1}(l_i + h_i) + \sum_{i=j}^{n-2} l_i = \sum_{i=0}^{n-2} l_i + \sum_{i=0}^{j-1} h_i$$
$$= y_0 + \sum_{i=0}^{j-1}(-y_i + y_{i+1}) = y_j = g(x).$$

□

4.2 Sigmoidal Activation Functions

Instead of step activation functions, standard network architectures use sigmoidal activation functions, which can be considered to be approximations of step functions. The reason for this is that standard training algorithms like backpropagation require differentiable activation functions.

In order to accommodate this, we will now approximate each step function s_i by a sigmoidal function σ_i:

$$\sigma_i(x) := \sigma_{l_i,h_i,m_i,z_i}(x) := l_i + \frac{h_i}{1 + e^{-z_i(x-m_i)}}.$$

Note that l_i, h_i, m_i are the same as for the step functions. The error of the i-th sigmoidal is

$$\delta_i(x) := |\sigma_i(x) - s_i(x)|.$$

An analysis of this function leads to the following results (illustrated in Figure 7): For all $x \neq m_i$ we have $\lim_{z_i \to \infty} \sigma_i(x) = s_i(x)$; since both functions are symmetric, we find for all $z_i, \Delta x$,

$$\delta_i(m_i - \Delta x) = \delta_i(m_i + \Delta x);$$

and furthermore, for all z_i, x, x' with $|x' - m_i| > |x - m_i|$,

$$\delta_i(x') < \delta_i(x).$$

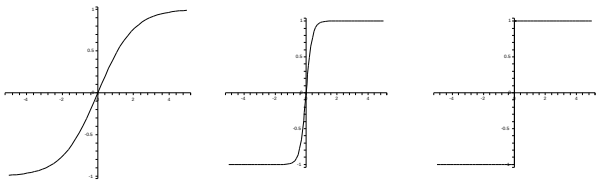

Figure 7: With increasing z, $\sigma_{l,h,m,z}$ gets arbitrarily close to $s_{l,h,m}$ everywhere but at m. The difference between $\sigma_{l,h,m,z}$ and $s_{l,h,m}$ is symmetric to m and decreases with increasing distance from m. Shown here are $\sigma_{-1,2,0,1}, \sigma_{-1,2,0,5}, s_{-1,2,0}$.

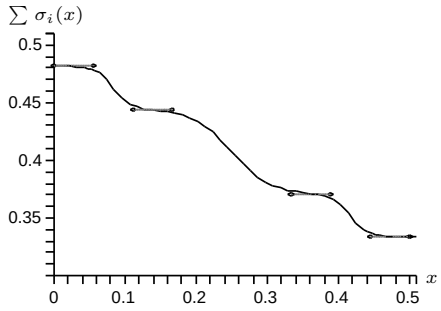

Figure 8: The sigmoidal approximation.

Theorem 4.3. For all $\epsilon > 0$ we can find z_i ($0 \leq i < n-1$) such that $\left|\sum_{i=0}^{n-2} \sigma_i(x) - g(x)\right| < \epsilon$.

Proof. In the worst case, the respective errors of the σ_i add up in the sum. Thus we allow a maximum error of $\epsilon' := \frac{\epsilon}{n-1}$ for each σ_i. With all previous results, it only remains to choose the z_i big enough to guarantee that at those $x \in D$ which are closest to m_i (i.e. c_i and a_{i+1}, which are equally close), σ_i approximates s_i up to ϵ', that is

$$\left[\delta_i(c_i) = \right] \delta_i(a_{i+1}) < \epsilon'.$$

Resolving this we get the following condition for the z_i:

$$z_i > \begin{cases} -\infty & \text{if } |h_i| \leq \epsilon' \\ -\frac{\ln \epsilon' - \ln(|h_i| - \epsilon')}{a_{i+1} - m_i} & \text{otherwise} \end{cases}$$

for $0 \leq i < n-1$. This completes the proof. □

Figure 8 shows the resulting sigmoidal approximation, along with the original piecewise constant function from Figure 6.

Taking g to be $\hat{f}_{P_l}$ and $\epsilon > 0$, the parameters l_i, h_i, m_i as in Definition 4.1 and z_i as in the proof of Theorem 4.3 determine an appropriate approximating sigmoidal network.

5 Constructing RBF Networks

Within the following section we will show how to construct *Radial Basis Function Networks* (RBF Networks). For a more detailed introduction for this type of network we refer to [14]. As in the previous section, we take a stepwise approach and will first discuss triangular activation functions.

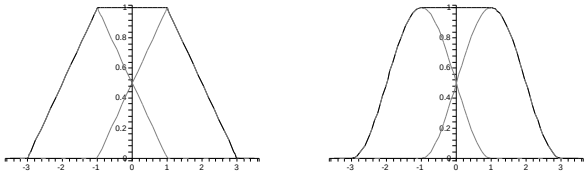

Figure 9: A constant piece can be obtained as the sum of two triangles or two raised cosine functions.

We will then extend the results to so-called raised cosine radial basis functions. We will also briefly discuss how an existing network can be refined incrementally to lower the error bound. The notation is the same as in the previous section. We will again assume that g is a piecewise constant function, this time with the additional requirement that the gaps between the pieces are $\geq$ the length of the pieces (which we proved to hold for $\hat{f}_{P_l}$), i.e. $c_i + b \leq a_{i+1}$ for $0 \leq i < n$.

5.1 Triangular Activation Functions

We will now construct an RBF network with distance input function, where each of the units in the hidden layer computes a triangular function $t_{w,h,m}$:

$$t_{w,h,m}(x) := \begin{cases} h \cdot \left(1 - \frac{|x-m|}{w}\right) & \text{if } |x-m| < w \\ 0 & \text{otherwise} \end{cases}$$

Since the triangular functions are locally receptive, that is, they are $\neq 0$ only on the open range $(m-w, m+w)$, we can handle each constant piece separately and represent it as a sum of two triangles, as illustrated in Figure 9.

For a given interval $[a_i, c_i]$ (with $c_i = a_i + b$), we define

$$t_i(x) := t_{b,y_i,a_i}(x), \qquad t'_i(x) := t_{b,y_i,c_i}(x).$$

Thus, for each constant piece we get two triangles summing up to that constant piece, i.e. for $0 \leq i < n$ and $x \in [a_i, c_i]$ we have $t_i(x) + t'_i(x) = y_i$, as illustrated in Figure 9.

The requirement we made for the gap between two constant pieces guarantees that the triangles do not interfere with those of other pieces.

Theorem 5.1. $\sum_{i=0}^{n-1} (t_i(x) + t'_i(x)) = g(x)$ for all $x \in D$.

Proof. This equality follows directly from the fact that the two triangles add up to a constant piece of the required height, and furthermore, that they do not interfere with other constant pieces as mentioned above. □

5.2 Raised-Cosine Activation Functions

As in the previous section, standard radial basis function network architectures use differentiable activation functions. For our purposes, we will replace the triangular functions t_i and t'_i by raised-cosine functions τ_i and τ'_i, respectively, of the following form:

$$\tau_{w,h,m}(x) := \begin{cases} \frac{h}{2} \cdot \left(1 + \cos\left(\frac{\pi(x-m)}{w}\right)\right) & \text{if } |x-m| < w \\ 0 & \text{otherwise.} \end{cases}$$

Again, we will use the following abbreviations:

$$\tau_i(x) := \tau_{b,y_i,a_i}(x) \qquad \tau'_i(x) := \tau_{b,y_i,c_i}(x)$$

As illustrated in Figure 9, raised cosines add up equally nice as the triangular ones, i.e. for $0 \leq i < n$ and $x \in [a_i, c_i]$ we have $\tau_i(x) + \tau'_i(x) = y_i$. Similar to Theorem 5.1, one easily obtains the following result.

Theorem 5.2. $\sum_{i=0}^{n-1} (\tau_i(x) + \tau'_i(x)) = g(x)$ for all $x \in D$.

As in the case of sigmoidal activation functions, we obtain the required network parameters by considering $\hat{f}_{P_l}$ instead of g.

5.3 Refining Networks

Our radial basis function network architecture lends itself to an incremental handling of the desired error bound. Assume we have already constructed a network approximating f_P up to a certain ϵ. We now want to increase the precision by choosing ϵ' with $\epsilon > \epsilon' > 0$, or by increasing the greatest relevant output level. Obviously we have $o_{\epsilon'} \geq o_\epsilon$ for $\epsilon > \epsilon' > 0$.

For this subsection, we have to go back to the original functions and domains from Section 3. Defining

$$\Delta P_{l_1,l_2} := \left\{ A \leftarrow \text{body} \in \mathcal{G}(P) \middle| l_1 < \|A\| \leq l_2 \right\},$$

one can easily obtain the following result.

Lemma 5.3. If $l_2 \geq l_1$, then $\hat{l}_2 \geq \hat{l}_1$, $D_{l_2} \subseteq D_{l_1}$, $P_{l_2} = P_{l_1} \cup \Delta P_{l_1,l_2}$, and $P_{l_1} \cap \Delta P_{l_1,l_2} = \emptyset$.

Thus, the constant pieces we had before may become divided into smaller pieces (if the greatest relevant input level increases) and may also be raised (if any of the new clauses applies to interpretations represented in the range of that particular piece).

Looking at the body atoms in $\Delta P_{l_1,l_2}$, we can identify the pieces which are raised, and then add units to the existing network which take care just of those pieces. Due to the local receptiveness of RBF units and the properties of D_l stated above, the new units will not disturb the results for other pieces. Especially in cases where $|\Delta P_{l_1,l_2}| \ll |P_{l_1}|$, this method may be more efficient than creating a whole new network from scratch.

We could also right away construct the network for P_l by starting with one for P_1 and refining it iteratively using $\Delta P_{1,2}, \Delta P_{2,3}, \ldots, \Delta P_{l-1,l}$, or maybe using difference programs defined in another way, e.g. by their greatest relevant input level. This may lead to more homogeneous constructions than the method used in the previous subsections.

6 Conclusions and Future Work

In this paper, we have shown how to construct connectionist systems which approximate covered first-order logic programs up to arbitrarily small errors, using some of the ideas proposed in [15]. We have thus, for a large class of logic programs, provided constructive versions of previous non-constructive existence proofs and extended previous constructive results for propositional logic programs to the first-order case.

An obvious alternative to our approach lies in computing the (propositional) ground instances of clauses of P up to a certain level and then using existing propositional constructions as in [11]. This approach was taken e.g. in [16], resulting in networks with increasingly large input and output layers. We avoided this for three reasons. Firstly, we want to obtain differentiable, standard architecture connectionist systems suitable for established learning algorithms. Secondly, we want to stay as close as possible to the first-order semantics in order to facilitate refinement and with the hope that this will make it possible to extract a logic program from a connectionist system. Thirdly, we consider it more natural to increase the number of nodes in the hidden layer for achieving higher accuracy, rather than to enlarge the input and output layers.

In order to implement our construction on a real computer, we are facing the problem that the hardware floating point precision is very limited, so we can only represent a small number of atoms in a machine floating point number. If we do not want to resort to programming languages emulating arbitrary precision, we could try to distribute the representation of interpretations on several units, i.e. to create a connectionist system with multi-dimensional input and output. For real applications, it would also be useful to further examine the possibilities for incremental refinement as in Section 5.3.

Another problem is that the derivative of the raised-cosine function is exactly 0 outside a certain range around the peak, which is not useful for training algorithms like backpropagation. Gaussian activation functions would be more suitable, but appear to be much more difficult to handle.

We are currently implementing the transformation algorithms, and will report on corresponding experiments on a different occasion. One of our long-term goals follows the path laid out in [7; 5] for the propositional case: to use logic programs as declarative descriptions for initialising connectionist systems, which can then be trained more quickly than randomly initialised ones, and then to understand the optimised networks by reading them back into logic programs.

References

[1] 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.

[2] Sebastian Bader, Artur S. d'Avila Garcez, and Pascal Hitzler. Computing first-order logic programs by fibring artificial neural networks. In *Proceedings of the 18th International FLAIRS Conference, Clearwater Beach, Florida, May 2005*, 2005. To appear.

[3] 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.

[4] 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.

[5] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125:155–207, 2001.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

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

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

[15] Anthony K. Seda. On the integration of connectionist and logic-based systems. In M. Schellekens T. Hurley, M. Mac an Airchinnigh and A. K. Seda, editors, *Proceedings of MFCSIT2004, Trinity College Dublin, July 2004*, Electronic Notes in Theoretical Computer Science, Elsevier, pages 1–24, 2005.

[16] Anthony K. Seda and Máire Lane. On approximation in the integration of connectionist and logic-based systems. In L. Li and K.K. Yen, editors, *Proceedings of the Third International Conference on Information, Tokyo, Japan*, pages 297–300. International Information Institute, 2004. ISBN 4-901329-02-2.