A Fully Connectionist Model Generator for Covered First-Order Logic Programs

Sebastian Bader[*] and **Pascal Hitzler**[†] and **Steffen Hölldobler**[*] and **Andreas Witzel**[‡]
[*] International Center for Computational Logic, Technische Universität Dresden, Germany
[†] AIFB, Universität Karlsruhe, Germany
[‡] Institute for Logic, Language and Computation, Universiteit van Amsterdam

Abstract

We present a fully connectionist system for the learning of first-order logic programs and the generation of corresponding models: Given a program and a set of training examples, we embed the associated semantic operator into a feed-forward network and train the network using the examples. This results in the learning of first-order knowledge while damaged or noisy data is handled gracefully.

1 Motivation

Three long-standing open research problems in connectionism are the questions of how to instantiate the power of symbolic computation within a fully connectionist system [Smolensky, 1987], how to represent and reason about structured objects and structure sensitive processes [Fodor and Pylyshyn, 1988], and how to overcome the propositional fixation [McCarthy, 1988], i.e. how to use connectionist systems for symbolic learning and reasoning beyond propositional logic. It has been shown that feed-forward networks are universal approximators and that artificial neural networks are Turing complete. Thus we know that symbolic computation is possible in principle, but at the same time the mentioned results are mainly theoretical.

Here we are concerned with the model generation for first-order logic programs, i.e. sets of rules which may contain variables ranging over infinite domains. Our approach is based on the following ideas first expressed in [Hölldobler *et al.*, 1999]: Various semantics of logic programs coincide with fixed points of associated semantic operators. Given that the semantic operator is continuous on the reals, the operator can be approximated arbitrarily well by a feed-forward network. In addition, if the operator is a contraction, then its fixed point can be computed by a recurrent extension of the feed-forward network.

Until now this approach was also purely theoretical for the first-order case. In this paper we show how feed-forward networks approximating the semantic operator of a given first-order logic program can be constructed, we show how these networks can be trained using input-output examples, and we demonstrate that the obtained connectionist system is robust against damage and noise. In particular, and after stating necessary preliminaries in Section 2, we make the following novel contributions in Section 3: We define a new multi-dimensional embedding of semantic operators into the reals, we construct a feed-forward network to approximate these operators and we present a new learning method using domain knowledge. The resulting system is evaluated in Section 4. Finally, we draw some conclusions and point out what needs to be done in the future in Section 5. For an overview of related work we refer to [d'Avila Garcez *et al.*, 2002] and [Bader and Hitzler, 2005].

2 Preliminaries

In this section, some preliminary notions from logic programming and connectionist systems are presented, along with the Core Method as one approach to integrate both paradigms.

2.1 First-Order Logic Programs

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

Example 1. *The following is a covered logic program which will serve as our running example.*

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

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

A *level mapping* is a function assigning a natural number $|A| \geq 1$ to each ground atom A. For negative ground literals we define $|\neg A| := |A|$. A logic program P is called *acyclic*

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

Example 2. *Consider the program from Example 1 and let s^n denote the n-fold application of s. With $|e(s^n(0))| := 2n+1$ and $|o(s^n(0))| := 2n+2$, we find that P is acyclic.*

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

Example 3. *For the program P from Example 1 we have $M := \{e(s^n(0)) \mid n \text{ even}\} \cup \{o(s^m(0)) \mid m \text{ odd}\} \models P$.*

Given a logic program P, the *single-step operator* $T_P : \mathcal{I}_\mathcal{L} \to \mathcal{I}_\mathcal{L}$ maps an interpretation I to the set of exactly those atoms A for which there is a clause $A \leftarrow \text{body} \in \mathcal{G}(P)$ such that the body is true under I. The operator T_P captures the semantics of P as the Herbrand models of the latter are exactly the pre-fixed points of the former, i.e. those interpretations I with $T_P(I) \subseteq I$. For logic programming purposes it is usually preferable to consider fixed points of T_P, instead of pre-fixed points, as the intended meaning of programs. These fixed points are called *supported models* of the program [Apt *et al.*, 1988]. In Example 1, the (obviously intended) model M is supported, while $\mathcal{B}_\mathcal{L}$ is a model but not supported.

Logic programming is an established and mature paradigm for knowledge representation and reasoning (see e.g. [Lloyd, 1988]) with recent applications in areas like rational agents or semantic web technologies (e.g. [Angele and Lausen, 2004]).

2.2 Connectionist Systems

A *connectionist system* is a network of simple computational units, 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-valued *weight*. Those units without incoming connections are called *input units*, those without outgoing ones are called *output units*.

We will consider 3-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 intermediate layer is called the *hidden layer*.

Each unit has an *input function* which uses the connections' weights to merge its inputs into one single value, and an *output function*. An example for a so-called *radial basis* input function is $(\vec{w}, \vec{x}) \mapsto \sqrt{\sum_{i=1}^{n}(x_i - w_i)^2}$, where the x_i are the inputs and the w_i are the corresponding weights. Possible output functions are the sigmoidal function ($x \mapsto \frac{1}{1+e^{-x}}$, for the hidden layer) and the identity ($x \mapsto x$, usually used in the output layer). If only one unit of a layer is allowed to output a value $\neq 0$, the layer implements a *winner-take-all* behavior.

Connectionist systems are successfully used for the learning of complex functions from raw data called training samples. Desirable properties include robustness with respect to damage and noise; see e.g. [Rojas, 1996] for details.

2.3 The Core Method

In [Hölldobler and Kalinke, 1994; Hitzler *et al.*, 2004] a method was proposed to translate a propositional logic program P into a neural network, such that the network will settle down in a stable state corresponding to a model of the program. To achieve this goal, the single-step operator T_P associated with P was implemented using a connectionist system. This general approach is nowadays called the *Core Method* [Bader and Hitzler, 2005].

In [Hölldobler *et al.*, 1999], the idea was extended to first-order logic programs: It was shown that the T_P-operator of acyclic programs can be represented as a continuous function on the real numbers. Exploiting the universal approximation capabilities of 3-layered feed-forward networks, it was shown that those networks can approximate T_P up to any given accuracy. However, no algorithms for the generation of the networks from given programs were presented. This was finally done in [Bader *et al.*, 2005] in a preliminary fashion.

3 The FineBlend System

In this section we will first discuss a new embedding of interpretations into vectors of real numbers. This extends the approach presented in [Hölldobler *et al.*, 1999] by computing m-dimensional vectors instead of a single real number, thus allowing for a higher and scalable precision. Afterwards, we will show how to construct a connectionist system approximating the T_P-operator of a given program P up to a given accuracy ε. As mentioned above, in [Bader *et al.*, 2005] first algorithms were presented. However, the accuracy obtainable in practice was limited through the use of a single real number for the embedding. The approach presented here allows for arbitrarily precise approximations. Additionally, we will present a novel training method, tailored for our specific setting. The system presented here is a fine blend of techniques from the *Supervised Growing Neural Gas (SGNG)* [Fritzke, 1998] and the approach presented in [Bader *et al.*, 2005].

3.1 Embedding

Obviously, we need to link the space of interpretations and the space of real vectors in order to feed the former into a connectionist system. To this end, we will first extend level mappings to a multi-dimensional setting, and then use them to represent interpretations as real vectors.

Definition 4. *An m-*dimensional level mapping *is a bijective function $\|\cdot\| : \mathcal{B}_\mathcal{L} \to (\mathbb{N}^+, \{1, \ldots, m\})$. For $A \in \mathcal{B}_\mathcal{L}$, if $\|A\| = (l, d)$, then l and d are called* level *and* dimension *of A, respectively. Again, we define $\|\neg A\| := \|A\|$.*

Definition 5. *Let $b \geq 3$ and let $A \in \mathcal{B}_\mathcal{L}$ be an atom with $\|A\| = (l, d)$. The* m-dimensional embedding *$\boldsymbol{\iota} : \mathcal{B}_\mathcal{L} \to \mathbb{R}^m$ and its extension $\boldsymbol{\iota} : \mathcal{I}_\mathcal{L} \to \mathbb{R}^m$ are defined as $\boldsymbol{\iota}(A) := (\iota_1(A), \ldots, \iota_m(A))$ where*

$$\iota_j(A) := \begin{cases} b^{-l} & \text{if } j = d \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \boldsymbol{\iota}(I) := \sum_{A \in I} \boldsymbol{\iota}(A).$$

With $\mathfrak{C}^m$ we denote the set of all embedded interpretations, i.e. $\mathfrak{C}^m := \{\boldsymbol{\iota}(I) \mid I \in \mathcal{I}_\mathcal{L}\} \subset \mathbb{R}^m$.[1]

[1] For $b = 2$, $\boldsymbol{\iota}$ is not injective, as $0.0\bar{1}_2 = 0.1_2$. We use $b = 4$.

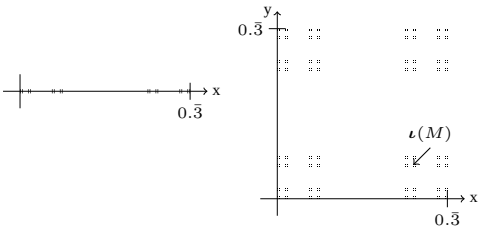

Figure 1: $\mathfrak{C}^1$ (left) and $\mathfrak{C}^2$ (right) for $b = 4$ and M from Ex. 6.

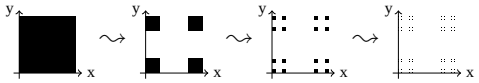

Figure 2: The first steps while constructing the limit $\mathfrak{C}^2$.

Example 6. *Using the 1-dimensional level mapping from Example 2, we obtain $\mathfrak{C}^1$ as depicted in Figure 1 on the left. Using the 2-dimensional level mapping $\|e(s^n(0))\| := (n+1, 1)$ and $\|o(s^n(0))\| := (n+1, 2)$, we obtain $\mathfrak{C}^2$ as depicted on the right and $\boldsymbol{\iota}(M) = (0.10\overline{10}_b, 0.01\overline{01}_b) \approx (0.2666667, 0.0666667)$ for the embedding of M.*

For readers familiar with fractal geometry, we note that $\mathfrak{C}^1$ is the classical Cantor set and $\mathfrak{C}^2$ the 2-dimensional variant of it [Barnsley, 1993]. Obviously, $\boldsymbol{\iota}$ is injective for a bijective level mapping and it is bijective on $\mathfrak{C}^m$. Using the m-dimensional embedding, the T_P-operator can be embedded into the real vectors to obtain a real-valued function f_P.

Definition 7. *The* m-dimensional embedding of T_P, *namely* $f_P : \mathfrak{C}^m \to \mathfrak{C}^m$, *is defined as* $f_P(\vec{x}) := \boldsymbol{\iota}\left(T_P\left(\boldsymbol{\iota}^{-1}(\vec{x})\right)\right)$.

The m-dimensional embedding of T_P is preferable to the one introduced in [Hölldobler *et al.*, 1999] and used in [Bader *et al.*, 2005], because it allows for scalable approximation precision on real computers. Otherwise, only 16 atoms could be represented with 32 bits.

Now we introduce *hyper-squares* which will play an important role in the sequel. Without going into detail, Figure 2 shows the first 4 steps in the construction of $\mathfrak{C}^2$. The big square is first replaced by 2^m shrunken copies of itself, the result is again replaced by 2^m smaller copies and so on. The limit of this iterative replacement is $\mathfrak{C}^2$. We will use $\mathfrak{C}^m_i$ to denote the result of the i-th replacement, i.e. Figure 2 depicts $\mathfrak{C}^2_0, \mathfrak{C}^2_1, \mathfrak{C}^2_2$ and $\mathfrak{C}^2_3$. Again, for readers with background in fractal geometry we note, that these are the first 4 applications of an iterated function system [Barnsley, 1993]. The squares occurring in the intermediate results of the constructions are referred to as *hyper-squares* in the sequel. H_l denotes a hyper-square of level l, i.e. one of the squares occurring in $\mathfrak{C}^m_l$. An approximation of T_P up to some level l will yield a function constant on all hyper-squares of level l.

Definition 8. *The* largest exclusive hyper-square *of a vector $\vec{u} \in \mathfrak{C}^m_0$ and a set of vectors $V = \{\vec{v}_1, \ldots, \vec{v}_k\} \subseteq \mathfrak{C}^m_0$, denoted by $H_{ex}(\vec{u}, V)$, either does not exist or is the hyper-square H of least level for which $\vec{u} \in H$ and $V \cap H = \emptyset$. The* smallest inclusive hyper-square *of a non-empty set of vectors $U = \{\vec{u}_1, \ldots, \vec{u}_k\} \subseteq \mathfrak{C}^m_0$, denoted by $H_{in}(U)$, is the hyper-square H of greatest level for which $U \subseteq H$.*

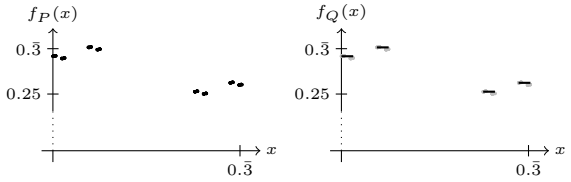

Figure 3: f_P for the program from Example 1 and the embedding from Example 2 is shown on the left. A piecewise constant approximation f_Q (level $l = 2$) is shown on the right.

3.2 Construction

In this section, we will show how to construct a connectionist network N for a given covered program P and a given accuracy ε, such that the dimension-wise maximum distance $d(f_P, f_N) := \max_{x,j}(|\pi_j(f_P(x)) - \pi_j(f_N(x))|)$ between the embedded T_P-operator f_P and the function f_N computed by N is at most ε. We will use a 3-layered network with a winner-take-all hidden layer.

With $l = \left\lceil \frac{-\ln((b-1)\varepsilon)}{\ln(b)} \right\rceil$, we obtain a level l such that whenever two interpretations I and J agree on all atoms up to level l in dimension j, we find that $|\iota_j(I) - \iota_j(J)| \leq \varepsilon$. For a covered program P, we can construct a finite subset $Q \subseteq \mathcal{G}(P)$ such that for all $I \in \mathcal{I}_\mathcal{L}$, $T_P(I)$ and $T_Q(I)$ agree on all atoms up to level l in all dimensions, hence $d(f_P, f_Q) \leq \varepsilon$. Furthermore, we find that the embedding f_Q is constant on all hyper-squares of level l [Bader *et al.*, 2005], i.e. we obtain a piecewise constant function f_Q such that $d(f_P, f_Q) \leq \varepsilon$.

We can now construct the feed-forward network as follows: For each hyper-square H of level l, we add a unit to the hidden layer, such that the input weights encode the position of the center of H. The unit shall output 1 if it is selected as winner, and 0 otherwise. The weight associated with the output connections of this unit is the value of f_Q on that hyper-square. Thus, we obtain a connectionist network approximating the semantic operator T_P up to the given accuracy ε. To determine the winner for a given input, we designed a locally receptive activation function such that its outcome is smallest for the closest "responsible" unit. Responsible units here are defined as follows: Given some hyper-square H, units which are positioned in H but not in any of its sub-hyper-squares are called *default units* of H, and they are responsible for inputs from H except for inputs from sub-hyper-squares containing other units. If H does not have any default units, the units positioned in its sub-hyper-squares are responsible for all inputs from H as well. When all units' activations have been (locally) computed, the unit with the smallest value is selected as the winner.

The following example is taken from [Witzel, 2006] and used to convey the underlying intuitions. All constructions work for m-dimensional embeddings in general, but for clarity the graphs here result from a 1-dimensional level mapping.

Example 9. *Using the program from Example 1 and the 1-dimensional level mapping from Example 2 we obtain f_P and f_Q for level $l = 2$ as depicted in Figure 3. The corresponding network consists of 1 input, 4 hidden and 1 output units.*

3.3 Training

In this section, we will describe the adaptation of the system during training, i.e. how the weights and the structure of a network are changed, given training samples with input and desired output, in such a way that the distribution underlying the training data is better represented by the network. This process can be used to refine a network resulting from an incorrect program, or to train a network from scratch. The training samples in our case come from the original (non approximated) program, but might also be observed in the real world or given by experts. First we discuss the adaptation of the weights and then the adaptation of the structure by adding and removing units. Some of the methods used here are adaptations of ideas described in [Fritzke, 1998]. For a more detailed discussion of the training algorithms and modifications we refer to [Witzel, 2006].

Adapting the weights Let $\vec{x}$ be the input, $\vec{y}$ be the desired output and u be the winner-unit from the hidden layer. To adapt the system, we change the output weights for u towards the desired output, i.e. $\vec{w}_{out} \leftarrow \eta \cdot \vec{y} + (1-\eta) \cdot \vec{w}_{out}$. Furthermore, we move u towards the center $\vec{c}$ of $H_{in}(\{\vec{x}, u\})$, i.e. $\vec{w}_{in} \leftarrow \mu \cdot \vec{c} + (1-\mu) \cdot \vec{w}_{in}$, where η and μ are predefined learning rates. Note that the winner unit is not moved towards the input but towards the center of the smallest hyper-square including the unit and the input. The intention is that units should be positioned in the center of the hyper-square for which they are responsible.

Adding new units The adjustment described above enables a certain kind of expansion of the network by allowing units to move to positions where they are responsible for larger areas of the input space. A refinement now should take care of densifying the network in areas where a great error is caused. Therefore, when a unit u is selected for refinement,[2] 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 containing u. Now u is moved to the center of this area, and some information gathered by u is used to determine a sub-hyper-square into whose center a new unit is placed, and to set up the output weights for the new unit.

Removing inutile units Each unit maintains a utility value, initially set to 1, which decreases over time and increases only if the unit contributes to the network's output.[3] If a unit's utility drops below a threshold, the unit will be removed.

3.4 Robustness

The described system is able to handle noisy data and to cope with damage. Indeed, the effects of damage to the system are quite obvious: If a hidden unit u fails, the receptive area is taken over by other units, thus only the specific results learned for u's receptive area are lost. While a corruption of the input weights may cause no changes at all in the network function, generally it can alter the unit's receptive area. If the output weights are corrupted, only certain inputs are effected. If the damage to the system occurs during training, it will be repaired very quickly as indicated by the experiment reported in Section 4.3. Noise is generally handled gracefully, because wrong or unnecessary adjustments or refinements can be undone in the further training process.

[2] The error for a given sample is ascribed to the winner unit. After a predefined number of training cycles, the unit with the greatest accumulated error is refined, if the error exceeds a given threshold.

[3] The contribution of a unit is the expected increase of error if the unit would be removed [Fritzke, 1998].

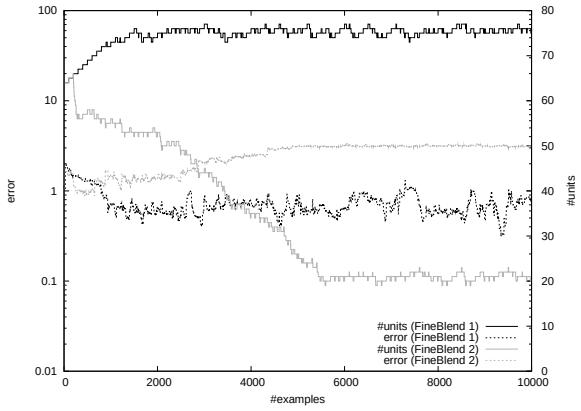

Figure 4: FineBlend 1 versus FineBlend 2.

4 Evaluation

In this section we will discuss some preliminary experiments. In the diagrams, we use a logarithmic scale for the error axis, and the error values are relative to ε, i.e. a value of 1 designates an absolute error of ε. For incorrect network initialization, we used the following wrong program:

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

Training samples were created randomly using the semantic operator of the program from Example 1.

4.1 Variants of Fine Blend

To illustrate the effects of varying the parameters, we use two setups: One with softer utility criteria (FineBlend 1) and one with stricter ones (FineBlend 2). Figure 4 shows that, starting from the incorrect initialization, the former decreases the initial error, paying with an increasing number of units, while the latter significantly decreases the number of units, paying with an increasing error. Hence, the performance of the network critically depends on the choice of the parameters. The optimal parameters obviously depend on the concrete setting, e.g. the kind and amount of noise present in the training data, and methods for finding them will be investigated in the future. For our further experiments we will use the FineBlend 1 parameters, which resulted from a mixture of intuition and (non-exhaustive) comparative simulations.

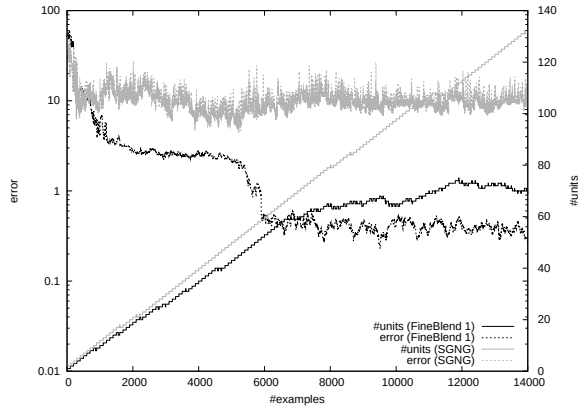

Figure 5: FineBlend 1 versus SGNG.

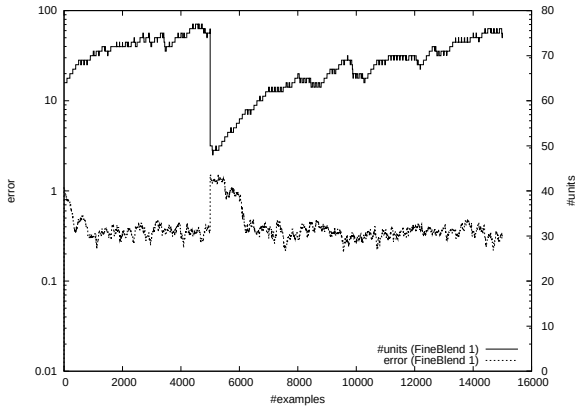

Figure 6: The effects of unit failure.

4.2 Fine Blend versus SGNG

Figure 5 compares FineBlend 1 with SGNG [Fritzke, 1998]. Both start off similarly, but soon SGNG fails to improve further. The increasing number of units is partly due to the fact that no error threshold is used to inhibit refinement, but this should not be the cause for the constantly high error level. The choice of SGNG parameters is rather subjective, and even though some testing was done to find them, they might be far from optimal. Finding the optimal parameters for SGNG is beyond the scope of this paper; however, it should be clear that it is not perfectly suited for our specific application. This comparison to an established generic architecture shows that our specialized architecture actually works, i.e. it is able to learn, and that it achieves the goal of specialization, i.e. it outperforms the generic architecture in our specific setting.

4.3 Unit Failure

Figure 6 shows the effects of unit failure. A FineBlend 1 network is (correctly) initialized and refined through training with 5000 samples, then one third of its hidden units are removed randomly, and then training is continued as if nothing had happened. The network proves to handle the damage gracefully and to recover quickly. The relative error exceeds 1 only slightly and drops back very soon; the number of units continues to increase to the previous level, recreating the redundancy necessary for robustness.

4.4 Iterating Random Inputs

One of the original aims of the Core Method is to obtain connectionist systems for logic programs which, when iteratively feeding their output back as input, settle to a stable state corresponding to an approximation of a fixed point of the program's single-step operator. In our running example, a unique fixed point is known to exist. To check whether our system reflects this, we proceed as follows:

1. Train a network from scratch until the relative error caused by the network is below 1, i.e. network outputs are in the ε-neighborhood of the desired output.
2. Transform the obtained network into a recurrent one by connecting the outputs to the corresponding inputs.

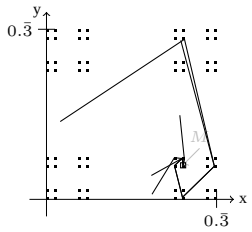

Figure 7: Iterating random inputs. The two dimensions of the input vectors are plotted against each other. The ε-neighborhood of the fixed point M is shown as a small box.

3. Choose a random input vector $\in \mathfrak{C}_0^m$ (which is not necessarily a valid embedded interpretation) and use it as initial input to the network.
4. Iterate the network until it reaches a stable state, i.e. until the outputs stay inside an ε-neighborhood.

For our example program, the unique fixed point of T_P is M as given in Example 3. Figure 7 shows the input space and the ε-neighborhood of M, along with all intermediate results of the iteration for 5 random initial inputs. The example computations converge, because the underlying program is acyclic [Witzel, 2006; Hölldobler *et al.*, 1999]. After at most 6 steps, the network is stable in all cases, in fact it is completely stable in the sense that all outputs stay exactly the same and not only within an ε-neighborhood. This corresponds roughly to the number of applications of our program's T_P operator required to fix the significant atoms, which confirms that the training method really implements our intention of learning T_P. The fact that even a network obtained through training from scratch converges in this sense further underlines the efficacy of our training method.

5 Conclusions and Further Work

We have reported on new results for overcoming the propositional fixation of current neural-symbolic systems: To the best of our knowledge this is the first constructive approach of approximating the semantic operators of first-order logic programs as well as their least fixed points in a fully connectionist setting. We also showed how the semantic operators

can be learned from given training examples using a modified neural gas method which exploits domain knowledge. The resulting system degrades gracefully under damage and noise, and recovers using training.

Whereas we define the embedding ι externally, in [Gust and Kühnberger, 2005] such embeddings are learned using ideas from category theory. In [Seda and Lane, 2005], connectionist systems for a covered program P are constructed by generating finite subsets of $\mathcal{G}(P)$ and employing the constructions presented in [Hölldobler and Kalinke, 1994].

Besides a thorough comparison of these approaches much remains to be done. The presented methods and procedures involve parameters, which are set manually; we would like to find (preferably optimal) parameters automatically. We would like to extract first-order logic programs after training, but all the extraction methods that we are aware of are propositional. This is a prerequisite not only to compare our method of learning semantic operators of logic programs with that of inductive logic programming, but also to complete the neural-symbolic learning cycle [Bader and Hitzler, 2005]. The investigation of realistic applications, e.g. to the learning of ontologies and other types of knowledge bases [Hitzler *et al.*, 2005] will follow.

Acknowledgments

We would like to thank three anonymous referees for their valuable comments on the preliminary version of this paper. 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 (grant 01 IMD01 B), and by the X-Media project (www.x-media-project.org) sponsored by the European Commission as part of the Information Society Technologies (IST) programme under EC grant number IST-FP6-026978. Andreas Witzel is supported by a Marie Curie Early Stage Research fellowship in the project GloRiClass (MEST-CT-2005-020841).

References

[Angele and Lausen, 2004] J. Angele and G. Lausen. Ontologies in F-Logic. In S. Staab and R. Studer, editors, *Handbook on Ontologies*, pages 29–50. Springer, 2004.

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

[Bader and Hitzler, 2005] S. Bader and P. Hitzler. Dimensions of neural-symbolic integration — a structured survey. In S. Artemov et al., editor, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 1, pages 167–194. King's College Publications, JUL 2005.

[Bader *et al.*, 2005] S. Bader, P. Hitzler, and A. Witzel. Integrating first-order logic programs and connectionist systems — a constructive approach. In A. S. d'Avila Garcez et al., editor, *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK*, 2005.

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

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

[Fodor and Pylyshyn, 1988] J. A. Fodor and Z. W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. In Pinker and Mehler, editors, *Connections and Symbols*, pages 3–71. MIT Press, 1988.

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

[Gust and Kühnberger, 2005] H. Gust and K.-U. Kühnberger. Learning symbolic inferences with neural networks. In B. Bara, L. Barsalou, and M. Bucciarelli, editors, *CogSci 2005: XXVII Annual Conference of the Cognitive Science Society*, pages 875–880, 2005.

[Hitzler *et al.*, 2004] P. Hitzler, S. Hölldobler, and A. K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 3(2):245–272, 2004.

[Hitzler *et al.*, 2005] P. Hitzler, S. Bader, and A. d'Avila Garcez. Ontology leaning as a use case for neural-symbolic integration. In A. Garcez et al., editor, *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy*, 2005.

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

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

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

[McCarthy, 1988] J. McCarthy. Epistemological challenges for connectionism. *Behavioural and Brain Sciences*, 11:44, 1988.

[Rojas, 1996] Raul Rojas. *Neural Networks*. Springer, 1996.

[Seda and Lane, 2005] Anthony K. Seda and Maire Lane. On approximation in the integration of connectionist and logic-based systems. In *Proceedings of the Third International Conference on Information (Information'04)*, pages 297–300, Tokyo, November 2005. International Information Institute.

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

[Witzel, 2006] A. Witzel. Neural-symbolic integration – constructive approaches. Master's thesis, Department of Computer Science, Technische Universität Dresden, Dresden, Germany, 2006.