The Core Method: Connectionist Model Generation

Sebastian Bader[⋆] and Steffen Hölldobler

International Center for Computational Logic
Technische Universität Dresden
01062 Dresden, Germany
Sebastian.Bader@inf.tu-dresden.de, sh@iccl.tu-dresden.de

Abstract. Knowledge based artificial networks networks have been applied quite successfully to propositional knowledge representation and reasoning tasks. However, as soon as these tasks are extended to structured objects and structure-sensitive processes it is not obvious at all how neural symbolic systems should look like such that they are truly connectionist and allow for a declarative reading at the same time. The core method aims at such an integration. It is a method for connectionist model generation using recurrent networks with feed-forward core. After an introduction to the core method, this paper will focus on possible connectionist representations of structured objects and their use in structure-sensitive reasoning tasks.

1 Introduction

From the very beginning artificial neural networks have been related to propositional logic. McCulloch-Pitts networks are finite automata and vice versa [22]. Finding a global minima of the energy function modelling a symmetric network corresponds to finding a model of a propositional logic formula and vice versa [23]. These are just two examples that illustrate what McCarthy has called a *propositional fixation* of connectionist systems in [21].

On the other hand, there have been numerous attempts to model first-order fragments in connectionist systems. In [3] energy minimization was used to model inference processes involving unary relations. In [19] and [27] multi-place predicates and rules over such predicates are modelled. In [16] a connectionist inference system for a limited class of logic programs was developed. But a deeper analysis of these and other systems reveals that the systems are in fact propositional. Recursive auto-associative memories based on ideas first presented in [25], holographic reduced representations [24] or the networks used in [9] have considerable problems with deeply nested structures. We are unaware of any connectionist system that fully incorporates structured objects and structure-sensitive processes and, thus, naturally incorporates the power of symbolic computation as argued for in e.g. [28].

⋆ The first author is supported by the GK334 of the German Research Foundation.

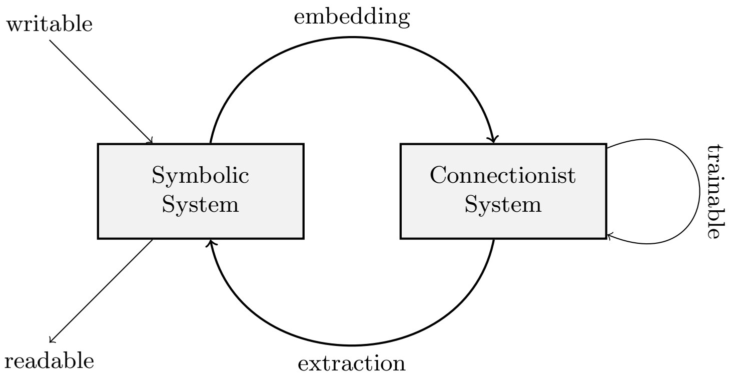

Fig. 1. The Neural-Symbolic Cycle

In this paper we are mainly interested in knowledge based artificial neural networks, i.e., networks which are initialized by available background knowledge before training methods are applied. In [29] it has been shown that such networks perform better than purely empirical and hand-built classifiers. [29] used background knowledge in the form of propositional rules and encodes these rules in multi-layer feed-forward networks. Independently, we have developed a connectionist system for computing the least model of propositional logic programs if such a model exists [14]. This system has been further developed to the so-called *core method*: background knowledge represented as logic programs is encoded in a feed-forward network, recurrent connections allow for a computation or approximation of the least model of the logic program (if it exists), training methods can be applied to the feed-forward kernel in order to improve the performance of the network, and, finally, an improved program can be extracted from the trained kernel closing the neural-symbolic cycle as depicted in Fig. 1.

In this paper we will present the core method in Section 3. In particular, we will discuss its propositional version including its relation to [29] and its extensions. The main focus of this paper will be on extending the core method to deal with structured objects and structure-sensitive processes in Section 4. In particular, we will give a feasability result, present a first practical implementation, and discuss preliminary experimental data. These main sections are framed by introducing basic notions and notations in Section 2 and an outlook in Section 5.

2 Preliminaries

We assume the reader to be familiar with basic notions from artificial neural networks and logic programs and refer to e.g. [4] and [20], resp. Nevertheless, we repeat some basic notions.

A *logic program* is a finite set of *rules* $H \leftarrow L_1 \wedge \cdots \wedge L_n$, where H is an atom and each L_i is a literal. H and $L_1 \wedge \cdots \wedge L_n$ are called the *post-* and *precondition* of

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

Fig. 2. A simple propositional logic program. The intended meaning of the rules is given on the right.

the rule, resp. Fig. 2 and 4 show a propositional and a first-order logic program, resp. These programs will serve as running examples. The knowledge represented by a logic program $\mathcal{P}$ can essentially be captured by the *meaning function* $T_{\mathcal{P}}$, which is defined as a mapping on the space of interpretations where for any interpretation I we have that $T_{\mathcal{P}}(I)$ is the set of all H for which there exists a ground instance $H \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg B_1 \wedge \cdots \wedge \neg B_n$ of a rule in $\mathcal{P}$ such that for all i we have $A_i \in I$ and for all j we have $B_j \notin I$, where each A_i and each B_j is an atom. Fixed points of $T_{\mathcal{P}}$ are called *(supported) models* of $\mathcal{P}$, which can be understood to represent the declarative semantics of $\mathcal{P}$.

Artificial neural networks consist of simple computational units (neurons), which receive real numbers as inputs via weighted connections and perform *simple* operations: the weighted inputs are added and simple functions (like threshold, sigmoidal) are applied to the sum. We will consider networks, where the units are organized in layers. Neurons which do not receive input from other neurons are called *input neurons*, and those without outgoing connections to other neurons are called *output neurons*. Such so-called *feed-forward networks* compute functions from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n and m are the number of input and output units, resp. Fig. 3 on the right shows a simple feed-forward network. In this paper we will construct recurrent networks by connecting the output units of a feed-forward network N to the input units of N. Fig. 3 on the left shows a blueprint of such a recurrent network.

3 The Core Method

In a nutshell, the idea behind the core method is to use feed-forward connectionist networks – called core – to compute or approximate the meaning function of logic programs. If the output layer of a core is connected to its input layer then these recurrent connections allow for an iteration of the meaning function leading to a stable state, corresponding to the least model of the logic program provided that such a least model exists (see Fig. 3 on the left). Moreover, the core can be trained using standard methods from connectionist systems. In other words, we are considering connectionst model generation using recurrent networks with feedforward core.

The ideas behind the core method were first presented in [14] for propositional logic programs (see also [13]). Consider the logic program shown in Fig. 2. A translation algorithm turns such a program into a core of logical threshold units. Because the program contains the predicate letters p, q and r only, it suffices

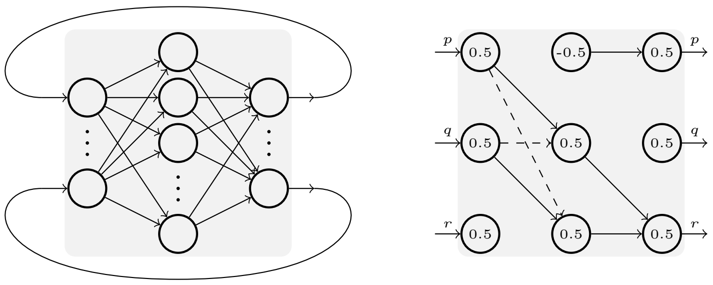

Fig. 3. The blueprint of a recurrent network used by the core method on the left. The core corresponding to $\mathcal{P}_1 = \{p,\ r \leftarrow p \wedge \neg q,\ r \leftarrow \neg p \wedge q\}$ is shown on the right. Solid connections have weight 1.0, dashed connections weight -1.0. The numbers within the units denote the thresholds.

to consider interpretations of these three letters. Such interpretations can be represented by triples of logical threshold units. The input and the output layer of the core consist exactly of such triples. For each rule of the program a logical threshold unit is added to the hidden layer such that the unit becomes active iff the preconditions of the rule are met by the current activation pattern of the input layer; moreover this unit activates the output layer unit corresponding to the postcondition of the rule. Fig. 3 on the right shows the network obtained by the translation algorithm if applied to $\mathcal{P}_1$.

In [14] we proved – among other results – that for each propositional logic program $\mathcal{P}$ there exists a core computing its meaning function $T_\mathcal{P}$ and that for each acyclic logic program $\mathcal{P}$ there exists a core with recurrent connections such that the computation with an arbitray intitial input converges and yields the unique fixed point of $T_\mathcal{P}$.

The use of logical threshold units in [14] made it easy to prove these results. However, it prevented the application of standard training methods like back-propagation to the kernel. This problem was solved in [8] by showing that the same results can be achieved if bipolar sigmoidal units are used instead (see also [5]). [8] also overcomes a restriction of the KBANN method originally presented in [29]: rules may now have arbitrarily many preconditions and programs may have arbitrarily many rules with the same postcondition.

In the meantime the propositional core method has been extended in many directions. In [18] three-valued logic programs are discussed; This approach has been extended in [26] to finitely determined sets of truth values. Modal logic programs have been considered in [6]. Answer set programming and metalevel priorities are discussed in [5]. The core method has been applied to intuitionistic logic programs in [7].

To summarize, the propositional core method allows for model generation with respect to a variety of logics in a connectionist setting. Given logic programs are translated into recurrent connectionist networks with feed-forward cores, such that the cores compute the meaning functions associated with the programs.

The cores can be trained using standard learning methods leading to improved logic programs. These improved programs must be extracted from the trained cores in order to complete the neural-symbolic cycle. The extraction process is outside the scope of this paper and interested readers are refered to e.g. [1] or [5].

4 The Core Method and Structured Objects

If structured objects and structure-sensitive processes are to be modelled, then usually higher-order logics are considered. In particular, first-order logic plays a prominent role because any computable function can be expressed by first-order logic programs. The extension of the core method to first-order logic poses a considerable problem because first-order interpretations usually do not map a finite but a countably infinite set of ground atoms to the set the truth values. Hence, they cannot be represented by a finite vector of units, each of which represents the value assigned to a particular ground atom.

In this section we will first show that an extension of the core method to first-order logic programs in feasible. However, the result will be purely theoretical and thus the question remains how cores can be constructed for first-order programs. In Subsection 4.2 a practical solution is discussed, which approximates the meaning functions of logic programs by means of piecewise constant functions. Some preliminary experimental data are presented in Subsection 4.3.

4.1 Feasibility

It is well known that multilayer feed-forward networks are universal approximators [17,12] of functions $\mathbb{R}^n \rightarrow \mathbb{R}^m$. Hence, if we find a way to represent interpretations of first-order logic programs by finite vector of real numbers, then feed-forward networks can be used to approximate the meaning function of such programs.

Consider a countably infinite set of ground atoms and assume that there is a bijection l uniquely assigning a natural number to each ground atom and vice versa; l is called *level mapping* and $l(A)$ *level* of the ground atom A. Furthermore, consider an interpretation I assigning to each ground atom A either 0 (representing falsehood) or 1 (representing truth) and let b be a natural number greater than 2. Then,

$$\iota(I) = \sum_{j=1}^{\infty} I(l^{-1}(j)) \cdot b^{-j},$$

is a real number encoding the interpretation I. With

$$\mathcal{D} = \{r \in \mathbb{R} \mid r = \sum_{j=1}^{\infty} a_j b^{-j}, a_j \in \{0,1\}\}$$

we find that ι is a bijection between the set of all interpretions and $\mathcal{D}$. Hence, we have a sound and complete encoding of interpretations.

Let $\mathcal{P}$ be a logic program and $T_{\mathcal{P}}$ its associated meaning operator. We define a sound and complete encoding $f_{\mathcal{P}} : \mathcal{D} \to \mathcal{D}$ of $T_{\mathcal{P}}$ as follows:

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

In [15] we proved – among other results – that for each logic program $\mathcal{P}$ which is acylic wrt. a bijective level mapping the function $f_{\mathcal{P}}$ is contractive, hence continuous. This has various implications: (i) We can apply Funahashi's result, viz. that every continuous function on (a compact subset of) the reals can be uniformly approximated by feed-forward networks with sigmoidal units in the hidden layer [12]. This shows that the meaning function of a logic program (of the kind discussed before) can be approximated by a core. (ii) Considering an appropriate metric, which will be discussed in a moment, we can apply Banach's contraction mapping theorem (see e.g. [30]) to conclude that the meaning function has a unique fixed point, which is obtained from an arbitrary initial interpretation by iterating the application of the meaning function. Using (i) and (ii) we were able to prove in [15] that the least model of logic programs which are acyclic wrt. a bijective level mapping can be approximated arbitrarily well by recurrent networks with feed-forward core.

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

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

As shown in [10] the set of all interpretations together with d is a complete metric space. Moreover, an interpretation I *approximates* an interpretation J *to degree* $n \in \mathbb{N}$ iff $d(I,J) \leq 2^{-n}$. In other words, if a recurrent network approximates the least model I of an acylic logic program to a degree $n \in \mathbb{N}$ and outputs $r \in \mathcal{D}$ then for all ground atoms A whose level is equal or less than n we find that $I(A) = \iota^{-1}(r)(A)$.

4.2 A First Approach

In this section, we will show how to construct a core network approximating the meaning operator of a given logic program. As above, we will consider logic programs $\mathcal{P}$ which are acyclic wrt. an bijective level mapping. We will construct sigmoidal networks and RBF networks with a raised cosine activation function. All ideas presented here can be found in detail in [2]. To illustrate the ideas, we will use the program $\mathcal{P}_2$ shown in Fig. 4 as a running example. The construction consists of five steps:

1. Construct $f_{\mathcal{P}}$.
2. Approximate $f_{\mathcal{P}}$ using a piecewise constant functions $\bar{f}_{\mathcal{P}}$.
3. Implement $\bar{f}_{\mathcal{P}}$ using (a) step and (b) triangular functions.

$$\begin{array}{lll} \mathcal{P}_2 = \{ & \mathit{even}(0). & \textit{\% 0 is an even number.} \\ & \mathit{even}(\mathit{succ}(X)) \leftarrow \mathit{odd}(X). & \textit{\% The successor of an odd X is even.} \\ & \mathit{odd}(X) \leftarrow \neg \mathit{even}(X).\ \} & \textit{\% If X is not even then it is odd.} \end{array}$$

Fig. 4. The first-order logic program $\mathcal{P}_2$ describing even and odd numbers. The intended meaning of the rules is given on the right.

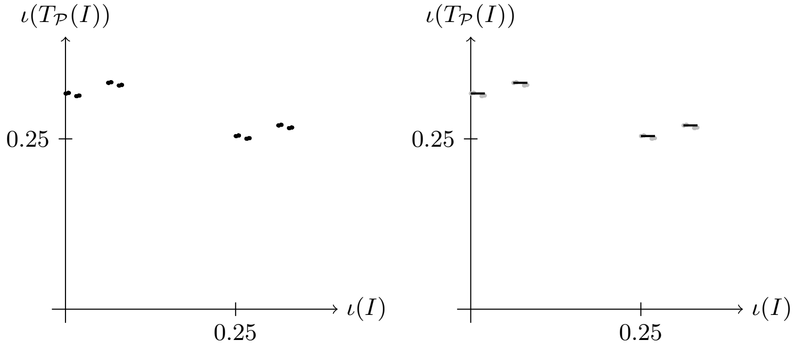

Fig. 5. On the left is the plot of $f_{\mathcal{P}_2}$. On the right a piecewise constant approximation $\bar{f}_{\mathcal{P}_2}$ (for level $n = 2$) of $f_{\mathcal{P}_2}$ is shown. The base $b = 4$ was used for the embedding.

4. Replace those by (a) sigmoidal and (b) raised cosine functions.
5. Construct the core network approximating $f_\mathcal{P}$.

In the sequel we will describe the ideas underlying the construction. A rigorous development including all proofs can be found in [2,31]. One should observe that $f_\mathcal{P}$ is a function on $\mathcal{D}$ and not on $\mathbb{R}$. Although the functions constructed below will be defined on intervals of $\mathbb{R}$, we are concerned with accuracy on $\mathcal{D}$ only.

1. Construct $f_\mathcal{P}$: $f_\mathcal{P}$ is defined as before, i.e., $f_\mathcal{P}(r) = \iota(T_\mathcal{P}(\iota^{-1}(r)))$. Fig. 5 on the left shows the plot of $f_{\mathcal{P}_2}$.

2. Constructing a Piecewise Constant Function $\bar{f}_\mathcal{P}$: Because $\mathcal{P}$ is acyclic, we conclude that all variables occurring in the precondition of a rule are also contained in its postcondition. Hence, for each level n we find that whenever $d(I, J) \leq 2^{-n}$ then $d(T_\mathcal{P}(I), T_\mathcal{P}(J)) \leq 2^{-n}$, where I and J are interpretations. Therefore, we can approximate $T_\mathcal{P}$ to degree n by some function $\bar{T}_\mathcal{P}$ which considers ground atoms with a level less or equal n only. As a consequence, we can approximate $f_\mathcal{P}$ by a piecewise constant function $\bar{f}_\mathcal{P}$ where each piece has a length of $\lambda = \frac{1}{(b-1)b^n}$, with b being the base used for the embedding. Fig. 5 shows $f_{\mathcal{P}_2}$ and $\bar{f}_{\mathcal{P}_2}$ for $n = 2$.

3. Implementation of $\bar{f}_\mathcal{P}$ using Linear Functions: As a next step, we will show how to implement $\bar{f}_\mathcal{P}$ using (a) step and (b) triangular functions. Those functions are the linear counterparts of the functions actually used in the networks

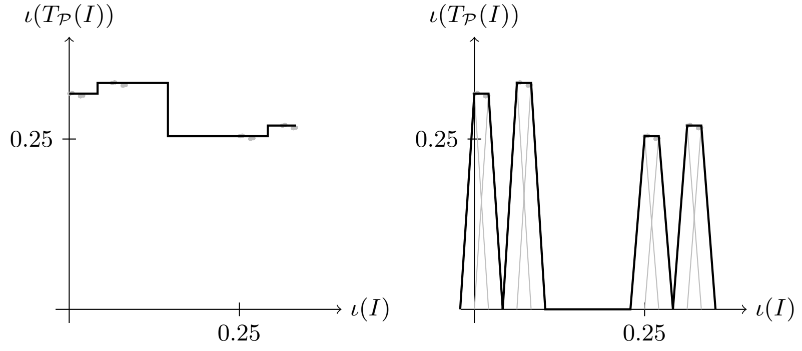

Fig. 6. Two linear approximation of $\bar{f}_{\mathcal{P}_2}$. On the left, three step functions were used; On the right, eight triangular functions (depicted in gray) add up to the approximation, which is shown using thick lines.

constructed below. If $\bar{f}_{\mathcal{P}}$ consists of k intervals, then we can implement it using $k-1$ step functions which are placed such that the steps are between two neighbouring intervals. This is depicted in Fig. 6 on the left.

Each constant piece of length λ could also be implemented using two triangular functions with width λ and centered at the endpoints. Those two triangles add up to the constant piece. For base $b = 4$, we find that the gaps between two intervals have a length of at least 2λ. Therefore, the triangular functions of two different intervals will never interfere. The triangular implementation is depicted in Fig. 6 on the right.

4. Implementation of $\bar{f}_{\mathcal{P}}$ using Nonlinear Functions: To obtain a sigmoidal approximation, we replace each step function with a sigmoidal function. Unfortunately, those add some further approximation error, which can be dealt with by increasing the accuracy in the constructions above. By dividing the desired accuracy by two, we can use one half as accuracy for the constructions so far and the other half as a margin to approximate the constant pieces by sigmoidal functions. This is possible because we are concerned with the approximation on $\mathcal{D}$ only.

The triangular functions described above can simply be replaced by raised cosine activation functions, as those add up exactly as the triangles do and do not interfere with other intervals either.

5. Construction of the Network: A standard sigmoidal core approximating the $T_{\mathcal{P}}$-operator of a given program $\mathcal{P}$ consists of:

- An input layer containing one input unit whose activation will represent an interpretation I.
- A hidden layer containing a unit with sigmoidal activation function for each sigmoidal function constructed above.
- An output layer containing one unit whose activation will represent the approximation of $T_{\mathcal{P}}(I)$.

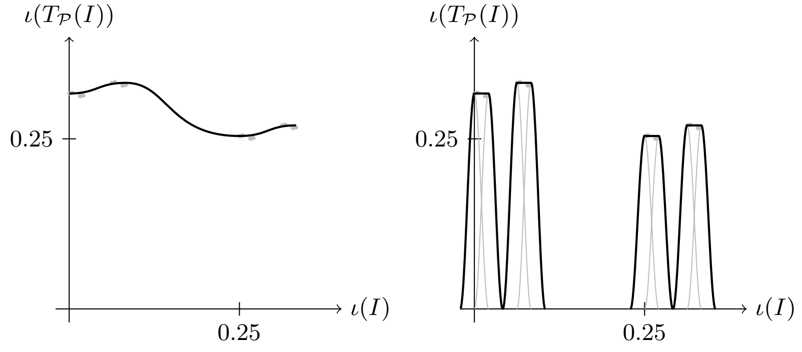

Fig. 7. Two non-linear approximation of $\bar{f}_{\mathcal{P}_2}$. On the left, sigmoidal functions were used and on the right, raised cosines.

The weights from input to hidden layer together with the bias of the hidden units define the positions of the sigmoidals. The weights from hidden to output layer represent the heights of the single functions. An RBF network can be constructed analogously, but will contain more hidden layer units, one for each raised cosine functions. Detailed constructions can be found in [2].

4.3 Evaluation and Experiments

In the previous section, we showed how to construct a core network for a given program and some desired level of accuracy. We used a one-dimensional embedding to obtain a unique real number $\iota(I)$ for a given interpretation I. Unfortunately, the precision of a real computer is limited, which implies, that using e.g. a 32-bit computer we could embed the first 16 atoms only. This limitation can be overcome by distributing an interpretation over more than one real number. In our running example $\mathcal{P}_2$, we could embed all *even*-atoms into one real number and all *odd*-atoms into another one, thereby obtaining a two-dimensional vector for each interpretation, hence doubling the accuracy. For various reasons, spelled out in [32], the sigmoidal approach described above does not work for more than one dimension. Nevertheless, an RBF network approach, similar to the one described above, does work. By embedding interpretations into higher-dimensional vectors, we can approximate meaning functions of logic programs arbitrarily well.

Together with some theoretical results, Andreas Witzel developed a prototype system in [32]. By adapting ideas from [11], he designed appropriate learning techniques utilizing the knowledge about a given domain, viz. the space of embedded interpretations. In the sequel, we will briefly present some of the results.

To adapt the networks behaviour during learning, the algorithm changes the weights, thereby changing the position and height of the constant pieces described above. Furthermore, new units are added if required, i.e., if a certain unit produces a large error, new units are added to support it. If a unit becomes inutile it will be removed from the network. These ideas are adaptations

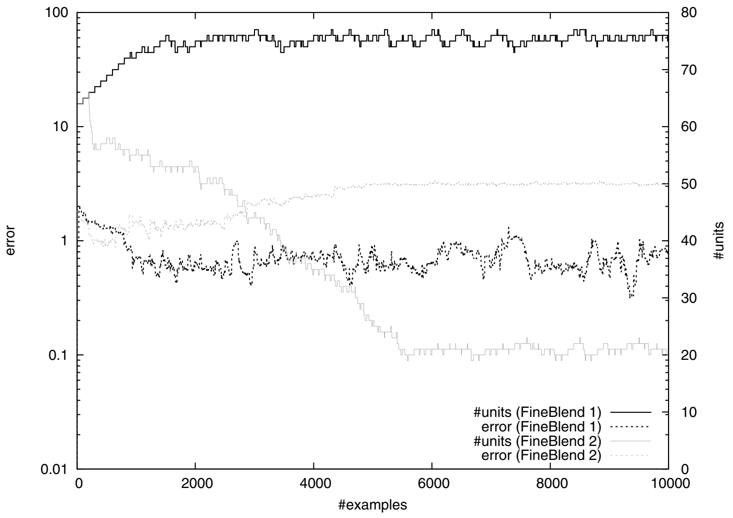

Fig. 8. Two different setups of the system during learning. Note that the error is shown on a logarithmic scale with respect to some given ε (1 means that the error is ε).

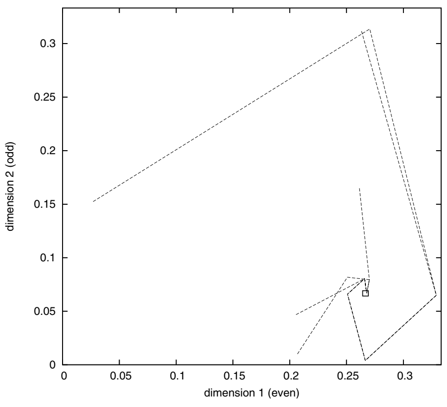

Fig. 9. Iterating random inputs

of concepts originally developed in the so called *growing neural gas* approach [11]. Fig. 8 shows a comparison of two different setups called FineBlend 1 and 2. FineBlend 1 is configured to keep the error below 1, whereas FineBlend 2 is configured to reduce the number of units resulting in a slightly higher error.

As mentioned above, a recurrent network is obtained by connecting output and input layer of the core. This is done to iterate the application of the meaning function. Therefore, we would assume a network set up and trained to represent the meaning function of an acyclic logic program to converge to a state representing the least model. As shown in Fig. 9, the network shows this behaviour.

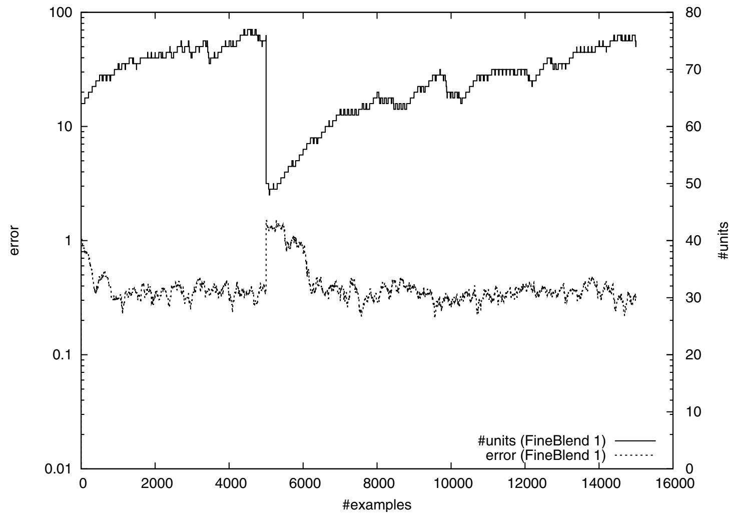

Fig. 10. The effect of unit failure. After 5000 examples, one third of the units were removed.

Shown are the two dimensions corresponding to the embedding of the even and odd predicates, resp. Also depicted is the ε-neighborhood of the least fixed point as a small square. Five random inputs were presented to the network and the output fed back via the recurrent connections. This process was repeated until the network reached a stable state, always being within the ϵ-neighbourhood of the fixed point.

Another advantage of connectionist systems is their robustness and their capability of repairing damage by further training. Fig. 10 shows the effect of unit failure. After presenting 5000 training samples to the network, one third of the hidden layer units were removed. As shown in the error plot, the system was able to recover quickly, thereby demonstrating its robustness. Further experiments and a more detailed analysis of the system can be found in [32,2].

5 Conclusion

We are currently implementing the first-order core method in order to further evaluate and test it using real world examples. Concerning a complete neural-symbolic cycle we note that whereas the extraction of propositional rules from trained networks is well understood, the extraction of first-order rules is an open question.

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

References

1. R. Andrews, J. Diederich, and A. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge–Based Systems*, 8(6), 1995.
2. S. Bader, P. Hitzler, and A. Witzel. Integrating first-order logic programs and connectionist systems — a constructive approach. In *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, Edinburgh, UK*, 2005.
3. D. H. Ballard. Parallel logic inference and energy minimization. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 203 – 208, 1986.
4. Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.
5. A.S. d'Avila Garcez, K. Broda, and D.M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications.* Springer, 2002.
6. A.S. d'Avila Garcez, L. C. Lamb, and D.M. Gabbay. A connectionist inductive learning system for modal logic programming. In *Proceedings of the IEEE International Conference on Neural Information Processing ICONIP'02*, Singapore, 2002.
7. A.S. d'Avila Garcez, L.C. Lamb, and D.M. Gabbay. Neural-symbolic intuitionistic reasoning. In *Design and Application of Hybrid Intelligent Systems*, pages 399–408, IOS Press, 2003.
8. A.S. d'Avila Garcez, G. Zaverucha, and L.A.V. de Carvalho. Logic programming and inductive learning in artificial neural networks. In Ch. Herrmann, F. Reine, and A. Strohmaier, editors, *Knowledge Representation in Neural Networks*, pages 33–46, Berlin, 1997. Logos Verlag.
9. J. L. Elman. Structured representations and connectionist models. In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 17–25, 1989.
10. M. Fitting. Metric methods – three examples and a theorem. *Journal of Logic Programming*, 21(3):113–127, 1994.
11. B. Fritzke. *Vektorbasierte Neuronale Netze.* Shaker Verlag, 1998.
12. K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.
13. P. Hitzler, S. Hölldobler, and A.K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.
14. S. Hölldobler and Y. Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.
15. S. Hölldobler, Y. Kalinke, and H.-P. Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–59, 1999.
16. S. Hölldobler and F. Kurfess. CHCL – A connectionist inference system. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, pages 318 – 342. Springer, LNAI *590*, 1992.
17. K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.
18. Y. Kalinke. Ein massiv paralleles Berechnungsmodell für normale logische Programme. Master's thesis, TU Dresden, Fakultät Informatik, 1994. (in German).
19. T. E. Lange and M. G. Dyer. High-level inferencing in a connectionist network. *Connection Science*, 1:181 – 217, 1989.
20. J. W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1993.

21. J. McCarthy. Epistemological challenges for connectionism. *Behavioural and Brain Sciences*, 11:44, 1988.
22. W. S. McCulloch and W. Pitts. A logical calculus and the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.
23. G. Pinkas. Symmetric neural networks and logic satisfiability. *Neural Computation*, 3:282–291, 1991.
24. T. A. Plate. Holographic reduced networks. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, 1992.
25. J. B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46:77–105, 1990.
26. A.K. Seda and M. Lane. Some aspects of the integration of connectionist and logic-based systems. In *Proceedings of the Third International Conference on Information*, pages 297–300, International Information Institute, Tokyo, Japan, 2004.
27. L. Shastri and V. Ajjanagadde. From associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioural and Brain Sciences*, 16(3):417–494, September 1993.
28. 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.
29. G.G. Towell and J.W. Shavlik. Extracting refined rules from knowledge–based neural networks. *Machine Learning*, 131:71–101, 1993.
30. S. Willard. *General Topology*. Addison–Wesley, 1970.
31. A. Witzel. Integrating first-order logic programs and connectionist networks. Project Thesis, Technische Universität Dresden, Informatik, 2005.
32. A. Witzel. Neural-symbolic integration – constructive approaches. Master's thesis, Technische Universität Dresden, Informatik, 2006.