Computing First-Order Logic Programs by Fibring Artificial Neural Networks

Sebastian Bader[*]
Department of Computer Science
Technische Universität Dresden
Germany

Artur S. d'Avila Garcez[†]
Department of Computing
City University London
UK

Pascal Hitzler[‡]
Institute AIFB
University of Karlsruhe
Germany

Abstract

The integration of symbolic and neural-network-based artificial intelligence paradigms constitutes a very challenging area of research. The overall aim is to merge these two very different major approaches to intelligent systems engineering while retaining their respective strengths. For symbolic paradigms that use the syntax of some first-order language this appears to be particularly difficult. In this paper, we will extend on an idea proposed by Garcez and Gabbay (2004) and show how first-order logic programs can be represented by fibred neural networks. The idea is to use a neural network to iterate a global counter n. For each clause C_i in the logic program, this counter is combined (fibred) with another neural network, which determines whether C_i outputs an atom of level n for a given interpretation I. As a result, the fibred network computes the single-step operator T_P of the logic program, thus capturing the semantics of the program.

Introduction

Intelligent systems based on artificial neural networks differ substantially from those based on symbolic knowledge processing like logic programming. Neural networks are trainable from raw data and are robust, but practically impossible to read declaratively. Logic programs can be implemented from problem specifications and can be highly recursive, while lacking good training methods and robustness, particularly when data are noisy (Thrun & others 1991). It is obvious that an integration of both paradigms into single systems would be very beneficiary if the respective strengths could be retained.

There exists a notable body of work investigating the integration of neural networks with propositional — or similarly finitistic — logic. We refer to (Browne & Sun 2001; d'Avila Garcez, Broda, & Gabbay 2002) for overviews. For first-order logic, however, it is much less clear how a reasonable integration can be achieved, and there are systematic difficulties which slow down recent research efforts, as spelled out in (Bader, Hitzler, & Hölldobler 2004). Different techniques for overcoming these obstacles are currently under investigation, including the use of metric spaces and topology, and of iterated function systems (Hitzler, Hölldobler, & Seda 2004; Bader & Hitzler 2004).

At the heart of these integration efforts is the question of how first-order knowledge can be represented by neural network architectures. In this paper, we present a novel approach using *fibring neural networks* as proposed by (d'Avila Garcez & Gabbay 2004). For each clause C_i of a logic program, a neural network that iterates a counter n is combined (fibred) with another neural network, which determines whether C_i outputs an atom of level n for a given interpretation I. Fibring offers a modular way of performing complex functions by using relatively simple networks (modules) in an ensemble.

The paper is organized as follows. In the next section we briefly review fibring neural networks and logic programs. We then present the fundamental ideas underlying our representation results, before giving the details of our implementation and a worked example. We conclude with some discussions.

Preliminaries

We introduce standard terminology for artificial neural networks, fibring neural networks, and logic programs. We refer the reader to (Bishop 1995; d'Avila Garcez & Gabbay 2004; Lloyd 1988), respectively, for further background.

Artificial Neural Networks

Artificial neural networks consist of simple computational units (neurons), which receive real numbers as inputs via weighted connections and perform *simple* operations: the weighted inputs are added and simple functions like threshold, sigmoidal, identity or truncate are applied to the sum.

The neurons are usually 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 output neurons. So a network computes a function from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n and m are the number of input, respectively, output units. A key to the success of

[*]Sebastian Bader is supported by the GK334 of the German Research Foundation.

[†]Artur Garcez is partly supported by The Nuffield Foundation.

[‡]Pascal Hitzler is supported by the German Federal Ministry of Education and Research under the SmartWeb project and by the European Union under the KnowledgeWeb Network of Excellence.

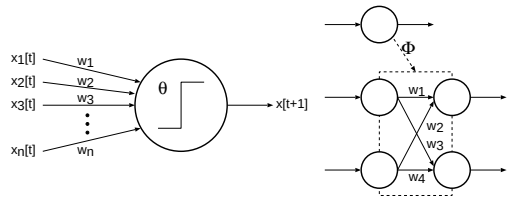


Figure 1: An artificial neuron (left) and a simple fibring network (right)

neural network architectures rests on the fact that they can be trained effectively using training samples in the form of input-output pairs.

For convenience, we make the following assumptions for the networks depicted in this paper: The layers are updated sequentially from left to right and within a layer the neurons are updated from top to bottom.

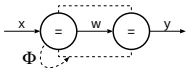
Recently, (d'Avila Garcez & Gabbay 2004) introduced a new model of neural networks, namely *fibring neural networks*. Briefly, the activation of a certain unit may influence the behaviour of other units by changing their weights. Our particular architecture is a slight variant of the original proposal, which appears to be more natural for our purposes.

Definition 1 *A fibring function Φ_i associated with neuron i maps some weights w of the network to new values, depending on w and the input x of neuron i.*

Fibring functions can be understood as modeling *presynaptic weights*, which play an important role in biological neural networks. Certainly, a necessary requirement for biological plausibility is that fibring functions compute either *simple* functions or tasks which can in turn be performed by neural networks. We will return to this point later.

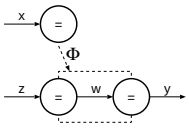
Throughout this paper we will use dashed lines, as in Figure 1, to indicate the weights which may be changed by some fibring function. As described above, we will use an update dynamics from left to right, and top to bottom. And, as soon as the activation of a fibring neuron is (re)calculated, the corresponding fibring function is applied and the respective weights are modified.

Example 2 A simple fibring network for squaring numbers. Each node computes the weighted sum of its inputs and performs the operation *identity* on it. The fibring function takes input x and multiplies it by W. If $W = 1$, the output will be $y = x^2$:



$$\Phi : (w, x) \mapsto x$$

Example 3 A simple fibring network implementing a gate-like behaviour. Nodes behave as in Example 2:



$$\Phi : (w, x) \mapsto \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The question of plausible types of fibring functions, as well as the computational power of those networks, will be studied separately and are touched here only slightly. We will start with very general fibring functions, but later we restrict ourselves to simple ones only, e.g. the fibred weight is simply multiplied by the activation.

Sometimes we will use the output of a neuron instead of the activation, or apply linear transformations to it, and it is clear that such modifications could also be achieved by adding another neuron to the network and use this for the fibring. Therefore these modifications can be understood as abbreviations to keep the networks simple.

Logic Programs

A *logic program* is a finite set of *clauses* $H \leftarrow L_1 \wedge \cdots \wedge L_n$, where $n \in \mathbb{N}$ may differ for each clause, H is an atom in a first order language $\mathcal{L}$ and $L_1, \ldots, L_n$ are literals, that is, atoms or negated atoms, in $\mathcal{L}$. The clauses of a program are understood as being universally quantified. H is called the *head* of the clause, each L_i is called a *body literal* and their conjunction $L_1 \wedge \cdots \wedge L_n$ is called the *body* of the clause. We allow $n = 0$, by an abuse of notation, which indicates that the body is empty; in this case the clause is called a *unit clause* or a *fact*.

An atom is said to be *ground* if it does not contain variables, and the *Herbrand base* underlying a given program $\mathcal{P}$ is defined as the set of all ground instances of atoms, denoted $B_\mathcal{P}$. Example 4 shows a logic program and its corresponding Herbrand base. Subsets of the Herbrand base are called (*Herbrand*) *interpretations* of $\mathcal{P}$, and we can think of such a set as containing those atoms which are *true* under the interpretation. The set $I_\mathcal{P}$ of all interpretations of a program $\mathcal{P}$ can thus be identified with the power set of $B_\mathcal{P}$.

Example 4 The *natural numbers* program $\mathcal{P}$, the underlying language $\mathcal{L}$ and the corresponding Herbrand base $B_\mathcal{P}$. The intended meaning of s is the successor function:

$\mathcal{P}$:	$nat(0).$ $nat(s(X)) \leftarrow nat(X).$
$\mathcal{L}$:	constants: $\mathcal{C} = \{0\}$ functions: $\mathcal{F} = \{s/1\}$ relations: $\mathcal{R} = \{nat/1\}$
$B_\mathcal{P}$:	$nat(0), nat(s(0)), nat(s(s(0))), \ldots$

Logic programs are accepted as a convenient tool for knowledge representation in logical form. Furthermore, the knowledge represented by a logic program $\mathcal{P}$ can essentially be captured by the *immediate consequence* or *single-step operator* $T_\mathcal{P}$, which is defined as a mapping on $I_\mathcal{P}$ where for any $I \in I_\mathcal{P}$ we have that $T_\mathcal{P}(I)$ is the set of all $H \in B_\mathcal{P}$ 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 clause in $\mathcal{P}$ such that for all i we have $A_i \in I$ and for all j we have $B_j \notin I$. Fixed points of $T_\mathcal{P}$ are called *supported models*

of P, which can be understood to represent the declarative semantics of P.

In the sequel of this paper we will often need to enumerate the Herbrand base, which is done via *level mappings*:

Definition 5 *Given a logic program P, a* level mapping *is a function $|\cdot| : B_{\mathcal{P}} \to \mathbb{N}^+$, where $\mathbb{N}^+$ denotes the set of positive integers excluding zero.*

Level mappings — in slightly more general form — are commonly used for controlling recursive dependencies between atoms, and the most prominent notion is probably the following.

Definition 6 *Let $\mathcal{P}$ be a logic program and $|\cdot|$ be a level mapping. If for all clauses $A \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_n \in ground(\mathcal{P})$ and all $1 \leq i \leq n$ we have that $|A| > |L_i|$, then $\mathcal{P}$ is called* acyclic *with respect to $|\cdot|$. A program is called* acyclic*, if there exists such a level mapping.*

Acyclic programs are known to have unique supported models (Cavedon 1991). The programs from Examples 4 and 7 below are acyclic.

Example 7 The "even and odd numbers" program and a level mapping:

$\mathcal{P}$:

$even(0).$
$even(s(X)) \leftarrow \neg even(X).$
$odd(s(X)) \leftarrow even(X).$

$|\cdot|$:

$$|A| = \begin{cases} 2 \cdot n + 1 & \text{if } A = even(s^n(0)) \\ 2 \cdot n + 2 & \text{if } A = odd(s^n(0)) \end{cases}$$

Throughout this paper we will assume that level mappings are bijective, i.e. for each $n \in \mathbb{N}^+$ there is exactly one $A \in B_{\mathcal{P}}$, such that $|A| = n$. Thus, for the purposes of our paper, a level mapping is simply an enumeration of the Herbrand base. Since level mappings induce an order on the atoms, we can use them to define a *prefix*-function on interpretations, returning only the first n atoms:

Definition 8 *The* prefix *of length n of a given interpretation I is defined as*

$$\begin{aligned} \mathrm{pref} &: I_{\mathcal{P}} \times \mathbb{N}^+ \to I_{\mathcal{P}} \\ &(I, n) \mapsto \{A | A \in I \text{ and } |A| \leq n\}. \end{aligned}$$

We will write $\mathrm{pref}_n(I)$ for $\mathrm{pref}(I, n)$.

For acyclic programs, it follows that in order to decide whether the atom with level $n+1$ must be included in $T_{\mathcal{P}}(I)$, it is sufficient to consider $\mathrm{pref}_n(I)$ only.

From Logic Programs to Fibring Networks

We will show how to represent acyclic logic programs by means of fibring neural networks. We follow up on the basic idea from (Hölldobler & Kalinke 1994; Hölldobler, Kalinke, & Störr 1999), and further developed in (Hitzler, Hölldobler, & Seda 2004; Bader & Hitzler 2004), to represent the single-step operator $T_{\mathcal{P}}$ by a network, instead of the program $\mathcal{P}$ itself. This is a reasonable thing to do since the single-step operator essentially captures the semantics of the program it is associated with, as mentioned before.

In order to represent $T_{\mathcal{P}}$ by the input-output mapping of a network, we also need an encoding of $I_{\mathcal{P}}$ as a suitable subset of the real numbers. We also use an idea from (Hölldobler, Kalinke, & Störr 1999) for this purpose. Let $B > 2$ be some integer, and let $|\cdot|$ be a bijective level mapping. Define

$$R : I_{\mathcal{P}} \to \mathbb{R} : I \mapsto \sum_{A \in I} B^{-|A|}.$$

We exclude $B = 2$, because in this case R would not be injective. It will be convenient to assume $B = 3$ throughout the paper, but our results do not depend on this. We denote the range of R by $\mathrm{range}(R)$.

There are systematic reasons why this way of embedding $I_{\mathcal{P}}$ into the reals is reasonable, and they can be found in (Hitzler, Hölldobler, & Seda 2004; Bader & Hitzler 2004), but will not concern us here. Using R, the prefix operation can be expressed naturally on the reals.

Proposition 9 *For $I \in I_{\mathcal{P}}$ and $x \in \mathrm{range}(R)$ we have*

$$\mathrm{pref}(I, n) = R^{-1}\left(\frac{\mathrm{trunc}(R(I) \cdot B^n)}{B^n}\right) \text{ and}$$

$$R(\mathrm{pref}(R^{-1}(x), n)) = \frac{\mathrm{trunc}(x \cdot B^n)}{B^n}.$$

For convenience, we overload pref and set $\mathrm{pref}(x, n) = R(\mathrm{pref}(R^{-1}(x), n))$ and $\mathrm{pref}_n(x) = \mathrm{pref}(x, n)$.

We will now turn to the construction of fibring networks which approximate given programs. We will first describe our approach in general terms, and spell it out in a more formal and detailed way later on. The goal is to construct a neural network, which will compute $R(T_{\mathcal{P}})(x) = R(T_{\mathcal{P}}(R^{-1}(x)))$ for a given $x \in \mathrm{range}(R)$. The network is designed in such a way that it successively approximates $R(T_P)(x)$ while running.

There will be a main loop iterating a global counter n. This counter fibres the kernel, which will evaluate whether the atom of level n is contained in $T_{\mathcal{P}}(I)$ or not, i.e. the kernel will output B^{-n} if the atom is contained, and 0 otherwise. Furthermore, there will be an input subnetwork providing $R(I)$ all the time, and the output subnetwork which will accumulate the outputs of the kernel, and hence converge to $R(T_{\mathcal{P}}(I))$.

For each clause C_i there is a subnetwork, which determines whether C_i outputs the atom of level n for the given interpretation I, or not. This is done by fibring the subnetwork such that it computes the corresponding ground instance $C_i^{(n)}$, with head of level n, if existent. If there is no

such ground instance, this subnetwork will output 0, otherwise it will determine whether the body is $true$ under the interpretation I. A detailed description of these clause networks will be given in the next section. Note that this construction is only possible for programs which are *covered*. This means that they do not have any local variables, i.e. every variable occuring in some body also occurs in the corresponding head. Obviously, programs which are acyclic with respect to a bijective level mapping are always covered.

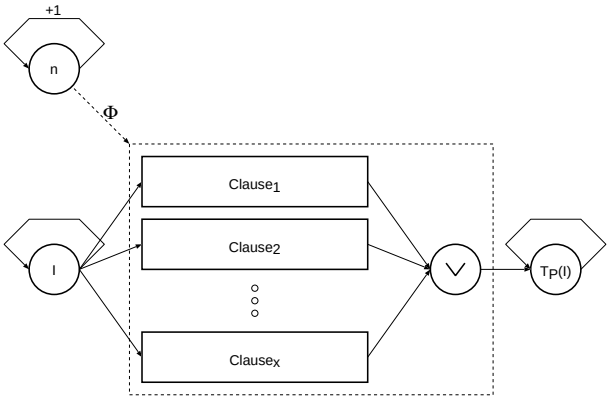


Figure 2: General architecture

If $\mathcal{P}$ is acyclic we can compute the unique supported model of the program directly, by connecting the output and the input region of the network as shown in Figure 3. This is simply due to the above mentioned fact: If we want to decide whether the atom of level n should be included in $T_{\mathcal{P}}(I)$, it is sufficient to look at the atoms $A \in I$ with level $< n$. We also have the following result.

Proposition 10 *Let $\mathcal{P}$ be a program which is acyclic with respect to a bijective level mapping $|\cdot|$, let $A \in B_{\mathcal{P}}$ with $|A| = n$. Then for each $I \in I_{\mathcal{P}}$ we have that $A \in T^n_{\mathcal{P}}(I)$ iff A is true with respect to the unique supported model of $\mathcal{P}$.*

Proof This is an immediate result from the application of the Banach contraction mapping principle to the semantic analysis of acyclic programs, see (Hitzler & Seda 2003). □

So, for acyclic programs, we can start with the empty (or any other) interpretation and let the (recurrent) network run.

Implementing Clauses

In order to complete the construction from the previous section, we give an implementation of the clauses. For a clause C of the form $H \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_k$, let $C^{(n)}$ denote the ground instance of C for which the head has level n, assuming it exists. The idea of the following construction is to create a network which implements C, and will be fibred by the counter n such that it implements $C^{(n)}$. In case that there is no ground instance of C with head of level n, the network will output 0, otherwise it will output 1 if the body is true with respect to the interpretation I, and 0 if it is not.

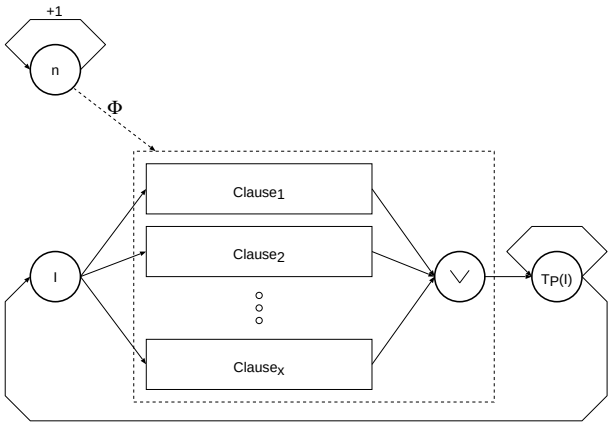


Figure 3: Recurrent architecture for acyclic programs

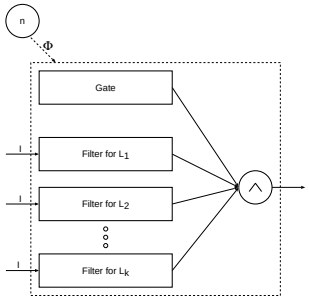


Figure 4: Implementing clauses

The idea, as shown in Figure 4, is that each subnetwork implementing a clause $C : H \leftarrow L_1 \wedge \ldots \wedge L_k$ with k body literals, consists of $k + 1$ parts — one *gate* and k *filters*. The gate will output 1, if the clause C has a ground instance $C^{(n)}$ where the level of the head is n. Furthermore there is a filter for each body literal L_i, which outputs 1, if the corresponding ground literal L_i is true under I. If all conditions are satisfied the final conjunction-neuron will become active, i.e. the subnetwork outputs 1.

Note that this construction again is sufficient only for programs which are covered. If we allowed local variables, then more than one (in fact infinitely many) ground instances of C with a head of level n could exist.

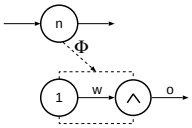
Let us have a closer look at the type of fibring function needed for our construction. For the gate, it implicitly performs a very *simple pattern matching* operation, checking whether the atom with level n unifies with the head of the clause. For the filters, it checks whether corresponding instances of body literals are true in the given interpretation, i.e. it implicitly performs a *variable binding* and an *elementary check of set-inclusion*.

We argue that the operations performed by the fibring function are indeed biologically feasible. The perspective which we take in this paper is that they should be understood as functions performed by a separate network, which we do not give explicitly, although we will substantiate this point to a certain extent in the next section. And pattern matching

is indeed a task that connectionist networks perform well. The variable binding task will also be addressed in the next section when we give examples for implementing the filters.

Neural Gates

As specified above, the gate for a clause $C : H \leftarrow L_1 \wedge \ldots \wedge L_k$ fires if there is a ground instance $C^{(n)}$ of C with head is of level n, as depicted in Figure 5. The decision based on



$$\Phi(w,n) = \begin{cases} 1 & \text{if ground instance with} \\ & \text{head of level } n \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

Figure 5: A neural gate

simple pattern matching is embedded into the fibring function. In what follows, we will discuss a number of different cases of how to unfold this fibring function into a network, in order to give plausible network topologies and yet simpler fibring functions. Other implementations are possible, and the cases presented here shall serve as examples only.

Ground-headed clauses. Let us first consider a clause for which the head does not contain variables, i.e. a ground clause, like for example the first clause given in Example 7 above. Since the level of the head in this case is fixed to some value, say m, the corresponding gate subnetwork should fire if and only if the general counter n is equal to m. This can be done using the network shown in Figure 6 (left): The neuron "1!" will always output 1 and the neuron "$= 0$" will output 1 if and only if the weighted inputs sum up to 0. This can easily be implemented using e.g. threshold units.

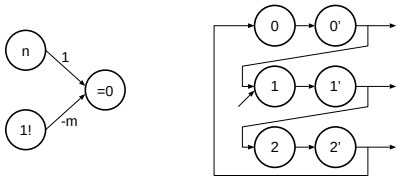
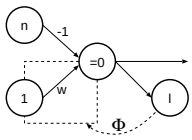


Figure 6: Simple gates for ground-headed clauses (left) and remainder classes (right)

Remainder classes. If the levels l_i of ground instantiated heads for a certain clause can be expressed as multiples of a certain fixed number m, i.e. $l_i = i \cdot m$ for all i (like clauses number 2 and 3 of Example 7), we can construct a simple subnetwork, as depicted in Figure 6 (right). The neurons symbolize the equivalence classes for the remainders of the devision by 3. The network will be initialized by activating "1". Every time it is reevaluated the activation simply proceeds to the next row.

Powers. If the level l_i of ground instantiated heads for a certain clause can be expressed as powers of a certain fixed number m, i.e. $l_i = m^i$ for all i, we can construct a simple subnetwork as shown in Figure 7.



$$w_{init} = -1$$
$$\Phi(w,x) = w \cdot m$$

Figure 7: A simple gate for powers

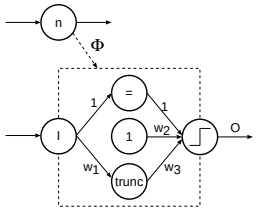
Filtering Interpretations

For a network to implement the ground instance $C^{(n)} : H_n \leftarrow L_1^{(n)} \wedge \ldots \wedge L_k^{(n)}$ of a clause C with head of level n, we need to know the *distance* between the head and the body literals — in terms of levels — as a function in n, i.e. we need a set of functions $\{b_i : \mathbb{N} \to \mathbb{N} \mid i = 1, \ldots, k\}$ — one for each body literal — where b_i computes the level of the literal L_i, taking as input the level of the head, as illustrated in Example 11.

Example 11 For the "even and odd numbers" program from Example 7, we can use the following b_i-functions:

$even(0).$	$\{\}$
$even(s(X)) \leftarrow \neg even(X).$	$\{b_1 : n \mapsto n-2\}$
$odd(s(X)) \leftarrow even(X).$	$\{b_1 : n \mapsto n-1\}$

For each body literal we will now construct a *filter* subnetwork, that fires if the corresponding ground body literal $L_i^{(n)}$ of $C^{(n)}$ is included in I. Given an interpretation I, we need to decide whether a certain atom A is included or not. The underlying idea is the following. In order to decide whether the atom A of level n is included in the interpretation I, we construct an interpretation J containing all atoms of I with level smaller than n, and the atom A, i.e. $J = \text{pref}_{n-1}(I) \cup \{A\}$, or, expressed on the reals, $R(J) = \text{pref}_{n-1}(R(I)) + B^{-n}$. If we evaluate $R(I) - R(J)$ the result will be non-negative if and only if A is included in I. This can be done using the network shown in Figure 8.



$$\Phi(w_1,n) = B^n$$
$$\Phi(w_2,n) = -B^{-n}$$
$$\Phi(w_3,n) = B^{-n}$$

Figure 8: Schematic plot and fibring function of a filter for the atom of level n

It is clear that we can construct networks to filter an atom of level $b_i(n)$, if the function b_i can itself be implemented in a neural network. Since fibring networks can implement any polynomial function, as shown in (d'Avila Garcez & Gabbay

2004) and indicated in Example 2, our approach is flexible and very general.

A Worked Example

Let us now give a complete example by extending on the logic program and the level mapping from Example 7 above. For the first clause we need a ground-headed gate only. To implement the second clause a remainder-class gate for the devision by 2 is needed, which returns 1 for all odd numbers. Furthermore, we need a filter which returns 1 if the atom of level $n-2$ is not included in I. For the last clause of the example, we need a gate returning 1 for all even numbers and a similar filter as for clause number 2. Combining all three parts and taking into account that $\mathcal{P}$ is acyclic, we get the network shown in Figure 9. If run on any initial value,

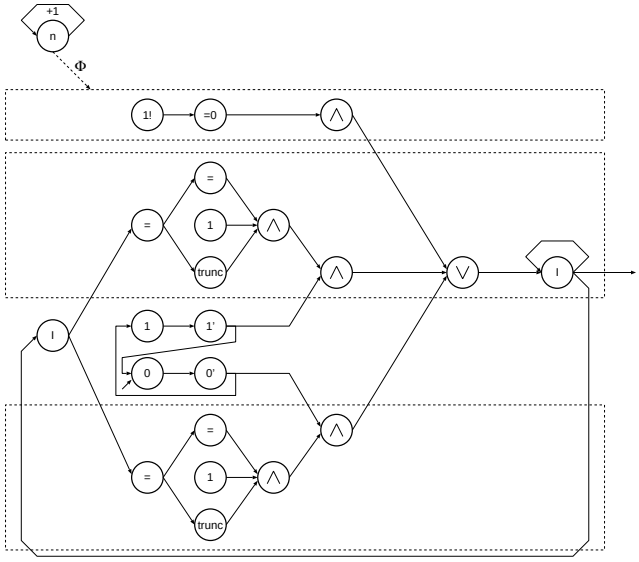


Figure 9: Neural implementation of the whole example

its outputs converge to the unique supported model of $\mathcal{P}$, i.e. the sequence of outputs of the right-most neuron is a sequence of real numbers which converges to $R(M)$, where M is the unique supported model of $\mathcal{P}$.

Conclusions

This paper contributes to advance the state of the art on neural-symbolic integration by showing how first-order logic programs can be implemented in fibring neural networks. Generic ways for representing the needed fibring functions in a biologically plausible fashion remain to be investigated in detail, as well as the task of extending our proposal towards a fully functional neural-symbolic learning and reasoning system.

Fibring offers a modular way of performing complex functions, such as logical reasoning, by combining relatively simple modules (networks) in an ensemble. If each module is kept simple enough, we should be able to apply standard neural learning algorithms to them. Ultimately, this may provide an integrated system with robust learning and expressive reasoning capability.

References

Bader, S., and Hitzler, P. 2004. Logic programs, iterated function systems, and recurrent radial basis function networks. *Journal of Applied Logic* 2(3):273–300.

Bader, S.; Hitzler, P.; and Hölldobler, S. 2004. The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. In *Proceedings of the Third International Conference on Information, Tokyo, Japan*. To appear.

Bishop, C. M. 1995. *Neural Networks for Pattern Recognition*. Oxford University Press.

Browne, A., and Sun, R. 2001. Connectionist inference models. *Neural Networks* 14(10):1331–1355.

Cavedon, L. 1991. Acyclic programs and the completeness of SLDNF-resolution. *Theoretical Computer Science* 86:81–92.

d'Avila Garcez, A. S., and Gabbay, D. M. 2004. Fibring neural networks. In McGuinness, D. L., and Ferguson, G., eds., *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, 342–347. AAAI Press / The MIT Press.

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

Hitzler, P., and Seda, A. K. 2003. Generalized metrics and uniquely determined logic programs. *Theoretical Computer Science* 305(1–3):187–219.

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

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

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

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

Thrun, S. B., et al. 1991. The MONK's problems: A performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, Carnegie Mellon University.