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 approximates 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 beneficial 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
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 $\Phi_i$ 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 $y = x^2$.

![Figure 1: An artificial neuron (left) and a simple fibring network (right)](image-url)

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 fibring 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 $H \leftarrow L_1 \land \cdots \land L_m$, where $n \in \mathbb{N}$ may differ for each clause, $H$ is an atom in a first order language $L$ and $L_1, \ldots, L_n$ are literals, that is, atoms or negated atoms, in $L$. 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 $L_1 \land \cdots \land L_n$ 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, literal or clause is said to be ground if it does not contain variables, and ground($\mathcal{P}$) denotes the ground instantiation of the program $\mathcal{P}$. 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 $L$ and the corresponding Herbrand base $B_\mathcal{P}$.

The intended meaning of $s$ is the successor function:

$$\mathcal{P} : \begin{align*}
\text{nat}(0). \\
\text{nat}(s(X)) & \leftarrow \text{nat}(X).
\end{align*}$$

$$L : \begin{align*}
\text{constants:} \quad & \mathcal{C} = \{0\} \\
\text{functions:} \quad & \mathcal{F} = \{s/1\} \\
\text{relations:} \quad & \mathcal{R} = \{\text{nat}/1\}
\end{align*}$$

$$B_\mathcal{P} : \begin{align*}
\text{nat}(0), \text{nat}(s(0)), \text{nat}(s(s(0))), & \ldots
\end{align*}$$

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 $H \leftarrow A_1 \land \cdots \land A_m \land \neg B_1 \land \cdots \land \neg B_n$ of a clause in $\mathcal{P}$ such that for all $i$ we have $A_i \in I$ and for all $j$ we have...
$B_i \notin I$. Fixed points of $T_P$ 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 $| \cdot | : B_P \to \mathbb{N}^+$, 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 $P$ be a logic program and $| \cdot |$ be a level mapping. If for all clauses $A \leftarrow L_1 \land L_2 \land \ldots \land L_n \in \text{ground}(P)$ and all $1 \leq i \leq n$ we have that $|A| > |L_i|$, then $P$ 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:

$P :$

- even(0).
- even(s(X)) ← ¬even(X).
- odd(s(X)) ← even(X).

$| \cdot | :$

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

Throughout this paper we will assume that level mappings are bijective, i.e. for each $n \in \mathbb{N}^+$ there is exactly one $A \in B_P$, 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 $x$ of length $n$ of a given interpretation $I$ is defined as

$\text{pref} : I_P \times \mathbb{N}^+ \to I_P$

$(I, n) \mapsto \{ A \mid A \in I \text{ and } |A| \leq n \}$.

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

For acyclic programs, it follows that in order to decide whether the atom with level $n + 1$ must be included in $T_P(I)$, it is sufficient to consider $\text{pref}_n(I)$ 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 $T_P$ by a network, instead of the program $P$ 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 $T_P$ by the input-output mapping of a network, we also need an encoding of $I_P$ 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

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

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 $\text{range}(R)$.

There are systematic reasons why this way of embedding $I_P$ 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 $I \in I_P$ and $x \in \text{range}(R)$ we have

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

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

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

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 $R(T_P(x))$ for a given $x \in \text{range}(R)$. 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 $T_P(I)$ 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 $R(T_P(I))$. Figure 2 shows the general architecture.

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 $C_i^{s(n)}$, with head of level $n$, if existent. If there is no
such ground instance, this subnetwork will output 0, other-
wise it will determine whether the body is true under the
interpretation $I$. A detailed description of these clause net-
works will be given in the next section. Note that this con-
struction is only possible for programs which are covered.
This means that they do not have any local variables, i.e. ev-
ery variable occurring in some body also occurs in the corre-
sponding head. Obviously, programs which are acyclic with
respect to a bijective level mapping are always covered.

If $P$ 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 $T_P(I)$, it
is sufficient to look at the atoms $A \in I$ with level $< n$. We
also have the following result.

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

**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 sec-
tion, we give an implementation of the clauses. For a clause
$C$ of the form $H \leftarrow L_1 \land L_2 \land \ldots \land L_k$, let $C^{(n)}$ denote
the ground instance of $C$ for which the head has level $n$, as-
suming it exists. The idea of the following construction is
to create a network which implements $C$, and will be fi-
bred 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.

The idea, as shown in Figure 4, is that each subnetwork
implementing a clause $C : H \leftarrow L_1 \land \ldots \land L_k$ 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

Note that this construction again is sufficient only for pro-
grams which are covered. If we allowed local variables, then
more than one (in fact infinitely many) ground instances of
$P$ which have 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 per-
forms 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 in-
stances of body literals are true in the given interpretation,
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 neuron “...” will output 1 if and only if the general counter clause, like for example the first clause given in Example 11.

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 \( C : H \leftarrow L_1 \land \ldots \land L_k \) fires if there is a ground instance \( C^{(n)} \) of \( C \) with head \( n \), as depicted in Figure 5. The decision based on

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

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 division 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.

**Filtering Interpretations**

For a network to implement the ground instance \( C^{(n)} : H_n \leftarrow L_1^{(n)} \land \ldots \land L_k^{(n)} \) 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:

\[
\begin{align*}
\text{even}(0), \\
\text{even}(s(X)) &\leftrightarrow \neg \text{even}(X). & \{ \} \\
\text{odd}(s(X)) &\leftrightarrow \text{even}(X). & \{ b_1 : n \mapsto n - 2 \}
\end{align*}
\]

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.

**Figure 7:** A simple gate for powers

\[
\Phi(w, x) = w \cdot m
\]

**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 division 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 \( P \) is acyclic, we get the network shown in Figure 9. If run on any initial value, its outputs converge to the unique supported model of \( P \), 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 \( P \).

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, including learning capabilities.

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


