Mathware & Soft Computing 12 (2005) 199-216

A Neural Implementation of Multi-Adjoint Logic Programs Via sf-homogenization

J. Medina, E. Mérida-Casermeiro, M. Ojeda-Aciego
Dept. Matemática Aplicada.
Universidad de Málaga*
{*jmedina,merida,aciego*}*@ctima.uma.es*

Abstract

A generalization of the homogenization process needed for the neural implementation of multi-adjoint logic programming (a unifying theory to deal with uncertainty, imprecise data or incomplete information) is presented here. The idea is to allow to represent a more general family of adjoint pairs, but maintaining the advantage of the existing implementation recently introduced in [6]. The soundness of the transformation is proved and its complexity is analysed. In addition, the corresponding generalization of the neural-like implementation of the fixed point semantics of multi-adjoint is presented.

1 Introduction

The study of reasoning methods under uncertainty, imprecise data or incomplete information has received increasing attention in the recent years. A number of different approaches have been proposed with the aim of better explaining observed facts, specifying statements, reasoning and/or executing programs under some type of uncertainty whatever it might be.

One important and powerful mathematical tool that has been used for this purpose at theoretical level is fuzzy logic. From the applicative side, neural networks have a massively parallel architecture-based dynamics which are inspired by the structure of human brain, adaptation capabilities, and fault tolerance. The recent paradigm of soft computing promotes the use and integration of different approaches for the problem solving.

The main advantages of fuzzy logic systems are the capability to express nonlinear input/output relationships by a set of qualitative if-then rules, and to handle both numerical data and linguistic knowledge, especially the latter, which is extremely difficult to quantify by means of traditional mathematics. The main advantage of neural networks, on the other hand, is the inherent learning capability, which enables the networks to adaptively improve their performance.

*Partially supported by TIC2003-09001-C02-01.

Recently, a new approach presented in [6] introduced a hybrid framework to handling uncertainty, expressed in the language of multi-adjoint logic but implemented by using ideas from the world of neural networks. The handling of uncertainty inside their logic model is based on the use of a generalised set of truth-values as a generalization of [11]. On the other hand, multi-adjoint logic programming [8] generalizes residuated logic programming [1] in that several different implications are allowed in the same program, as a means to facilitate the task of specification.

Considering several implications in the same program is interesting because it provides a more flexible framework for the specification of problems, for instance, in situations in which connectives are built from the users preferences. In these contexts, it is likely that knowledge is described by a many-valued logic program where connectives have many-valued truth functions and, perhaps, aggregation operators (such as arithmetic mean or weighted sum) where different implications could be needed for different purposes, and different aggregators are defined for different users, depending on their preferences.

The neural-like implementation of multi-adjoint logic programming in [6] had the restriction that the only connectives involved in the program were the usual product, Gödel and Łukasiewicz together with weighted sums. A key point of the implementation was a preprocessing of the initial program to transform it into a *homogeneous* program. As the theoretical development of the multi-adjoint framework does not rely on particular properties of the product, Gödel and Łukasiewicz adjoint pairs, it seems convenient to allow for a generalization of the implementation to admit, at least, a family of continuous t-norms (recall that any continuous t-norm can be interpreted as the ordinal sum of product and Łukasiewicz t-norms).

The authors presented in [7] a different neural approach allowing for ordinal sums in the initial program, which uses the same homogenization process introduced in [6]. The purpose of this paper is to present a "finer" homogenization process for multi-adjoint logic programs such that conjunctors which are built as ordinal sums of product, Gödel and Łukasiewicz t-norms are decomposed into its components so that, as a result, the original neural approach is still applicable.

The structure of the paper is as follows: In Section 2, the syntax and semantics of multi-adjoint logic programs are introduced, together with the homogenization procedure; in Section 3, the translation from multi-adjoint programs into homogeneous programs is extended in order to cope with conjunctors defined as ordinal sums, the preservation of the semantics is proved under this extended transformation, and the complexity of the translation is studied. Section 4 introduces the modification of the neural net of [6] in order to include the modification of the homogenization process and the comparison with the sequence of iterations of the $T_{\mathbb{P}}$ operator is presented; then, some comments about related approaches are given in Section 5. The paper finishes with some conclusions and pointers to future work.

2 Preliminary definitions

To make this paper as self-contained as possible, the necessary definitions about multi-adjoint structures are included in this section. For motivating comments, the

interested reader is referred to [8].

Multi-adjoint logic programming is a general theory of logic programming which allows the simultaneous use of different implications in the rules and rather general connectives in the bodies.

The first interesting feature of multi-adjoint logic programs is that a number of different implications are allowed in the bodies of the rules. The basic definition is the generalization of residuated lattice given below:

Definition 1 *A* multi-adjoint lattice $\mathcal{L}$ *is a tuple* $(L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ *satisfying the following items:*

1. $\langle L, \preceq \rangle$ *is a bounded lattice, i.e. it has bottom and top elements;*
2. $\top \,\&_i\, \vartheta = \vartheta \,\&_i\, \top = \vartheta$ *for all* $\vartheta \in L$ *for* $i = 1, \ldots, n$*;*
3. $(\&_i, \leftarrow_i)$ *is an* adjoint pair *in* $\langle L, \preceq \rangle$ *for* $i = 1, \ldots, n$*; i.e.*
   (a) *Operation* $\&_i$ *is increasing in both arguments,*
   (b) *Operation* $\leftarrow_i$ *is increasing in the first argument and decreasing in the second argument,*
   (c) *For any* $x, y, z \in P$*, we have that* $x \preceq (y \leftarrow_i z)$ *holds if and only if* $(x \,\&_i\, z) \preceq y$ *holds.*

In the rest of the paper we restrict to the unit interval, although the general framework of multi-adjoint logic programming is applicable to a general lattice.

2.1 Syntax of multi-adjoint logic programs

A *multi-adjoint program* is a set of weighted rules $\langle F, \vartheta \rangle$ satisfying the following conditions:

1. F is a formula of the form $A \leftarrow_i \mathcal{B}$ where A is a propositional symbol called the *head* of the rule, and $\mathcal{B}$ is a well-formed formula, which is called the *body*, built from propositional symbols $B_1, \ldots, B_n$ $(n \geq 0)$ by the use of monotone operators.
2. The *weight* ϑ is an element (a truth-value) of $[0, 1]$.

Facts are rules with body[1] 1 and a *query* (or *goal*) is a propositional symbol intended as a question $?A$ prompting the system.

2.2 Semantics of multi-adjoint logic programs

Once presented the syntax of multi-adjoint programs, the semantics is given below.

Definition 2 *An* interpretation *is a mapping* I *from the set of propositional symbols* Π *to the lattice* $\langle [0, 1], \leq \rangle$.

[1]It is also customary to use write $\top$ instead of 1, and even not to write any body.

Note that each of these interpretations can be uniquely extended to the whole set of formulas, and this extension is denoted as $\hat{I}$. The set of all the interpretations is denoted $\mathcal{I}_{\mathfrak{L}}$.

The ordering $\leq$ of the truth-values L can be easily extended to $\mathcal{I}_{\mathfrak{L}}$, which also inherits the structure of complete lattice and is denoted $\sqsubseteq$. The minimum element of the lattice $\mathcal{I}_{\mathfrak{L}}$, which assigns 0 to any propositional symbol, will be denoted $\triangle$.

Definition 3

1. *An interpretation* $I \in \mathcal{I}_{\mathfrak{L}}$ satisfies $\langle A \leftarrow_i \mathcal{B}, \vartheta \rangle$ *if and only if* $\vartheta \leq \hat{I}\,(A \leftarrow_i \mathcal{B})$.
2. *An interpretation* $I \in \mathcal{I}_{\mathfrak{L}}$ *is a* model *of a multi-adjoint logic program* $\mathbb{P}$ *iff all weighted rules in* $\mathbb{P}$ *are satisfied by* I.

The operational approach to multi-adjoint logic programs used in this paper will be based on the fixpoint semantics provided by the immediate consequences operator, given in the classical case by van Emden and Kowalski [10], which can be generalised to the framework of multi-adjoint logic programs by means of the adjoint property, as shown below:

Definition 4 *Let* $\mathbb{P}$ *be a multi-adjoint program; the* immediate consequences operator*,* $T_{\mathbb{P}} \colon \mathcal{I}_{\mathfrak{L}} \to \mathcal{I}_{\mathfrak{L}}$*, maps interpretations to interpretations, and for* $I \in \mathcal{I}_{\mathfrak{L}}$ *and* $A \in \Pi$ *is given by*

$$T_{\mathbb{P}}(I)(A) = \sup \Big\{ \vartheta \,\&_i\, \hat{I}(\mathcal{B}) \mid \langle A \leftarrow_i \mathcal{B}, \vartheta \rangle \in \mathbb{P} \Big\}$$

As usual, it is possible to characterise the semantics of a multi-adjoint logic program by the post-fixpoints of $T_{\mathbb{P}}$; that is, an interpretation I is a model of a multi-adjoint logic program $\mathbb{P}$ iff $T_{\mathbb{P}}(I) \sqsubseteq I$. The $T_{\mathbb{P}}$ operator is proved to be monotonic and continuous under very general hypotheses.

Once one knows that $T_{\mathbb{P}}$ can be continuous under very general hypotheses [8], then the least model can be reached in at most countably many iterations beginning with the least interpretation, that is, the least model is $T_{\mathbb{P}} \uparrow \omega(\triangle)$.

2.3 Obtaining a homogeneous program

Regarding the implementation as a neural network of [6], the introduction of the so-called *homogeneous rules*, provided a simpler and standard representation for any multi-adjoint program.

Definition 5 *A weighted formula is said to be* homogeneous *if it has one of the following forms:*

- $\langle A \leftarrow_i \&_i(B_1, \ldots, B_n), \vartheta \rangle$
- $\langle A \leftarrow_i @(B_1, \ldots, B_n), 1 \rangle$
- $\langle A \leftarrow_i B_1, \vartheta \rangle$

where $A, B_1, \ldots, B_n$ are propositional symbols.

In the rest of this section we briefly recall the procedure for translating a multi-adjoint logic program into one containing only homogeneous rules. The procedure handles separately rules and facts, the latter are not related to the purpose of this paper, therefore we will only recall the procedure presented for homogenizing rules.

Two types of transformations are considered: The first one handles the main connective of the body of the rule, whereas the second one handles the subcomponents of the body.

*T*1. A weighted rule $\langle A \leftarrow_i \&_j(\mathcal{B}_1, \ldots, \mathcal{B}_n), \vartheta\rangle$ is substituted by the following pair of formulas:

$$\begin{aligned} \langle A &\leftarrow_i A_1, \vartheta\rangle \\ \langle A_1 &\leftarrow_j \&_j(\mathcal{B}_1, \ldots, \mathcal{B}_n), 1\rangle \end{aligned}$$

where A_1 is a fresh propositional symbol, and $\langle \leftarrow_j, \&_j\rangle$ is an adjoint pair.

For the case $\langle A \leftarrow_i @(\mathcal{B}_1, \ldots, \mathcal{B}_n), \vartheta\rangle$ in which the main connective of the body of the rule happens to be an aggregator, the transformation is similar:

$$\begin{aligned} \langle A &\leftarrow_i A_1, \vartheta\rangle \\ \langle A_1 &\leftarrow @(\mathcal{B}_1, \ldots, \mathcal{B}_n), 1\rangle \end{aligned}$$

where A_1 is a fresh propositional symbol, and $\leftarrow$ is a designated implication.

*T*2. A weighted rule $\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_n), \vartheta\rangle$, where Θ is either $\&_i$ or an aggregator, and a component $\mathcal{B}_k$ is assumed to be either of the form $\&_j(\mathcal{C}_1, \ldots, \mathcal{C}_l)$ or $@(\mathcal{C}_1, \ldots, \mathcal{C}_l)$, is substituted by the following pair of formulas in either case:

$$\begin{aligned} \langle A &\leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_{k-1}, A_1, \mathcal{B}_{k+1}, \ldots, \mathcal{B}_n), \vartheta\rangle \\ \langle A_1 &\leftarrow_j \&_j(\mathcal{C}_1, \ldots, \mathcal{C}_l), 1\rangle \end{aligned}$$

or

$$\begin{aligned} \langle A &\leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_{k-1}, A_1, \mathcal{B}_{k+1}, \ldots, \mathcal{B}_n), \vartheta\rangle \\ \langle A_1 &\leftarrow @(\mathcal{C}_1, \ldots, \mathcal{C}_l), 1\rangle \end{aligned}$$

where A_1 is a fresh propositional symbol.

The procedure to transform the rules of a program so that all the resulting rules are homogeneous, is presented in Fig. 1. It is based in the two previous transformations, and in its description by abuse of notation the terms T1-rule (resp. T2-rule) are used to mean an adequate input rule for transformation T1 (resp. T2).

```
Program Homogenization
  begin
    repeat
      for each T1-rule do
        Apply transformation T1
      end-for
      for each T2-rule do
        Apply transformation T2
      end-for
    until neither T1-rules nor T2-rules exist
  end
```

Figure 1: Pseudo-code for translating into a homogeneous program.

3 Considering compound conjunctors

As stated in the introduction, a neural net implementation of the immediate consequences operator of an homogeneous program was introduced in [6] for the case of the multi-adjoint lattice $([0,1], \leq, \&_P, \leftarrow_P, \&_G, \leftarrow_G, \&_L, \leftarrow_L)$. As the theoretical development of the multi-adjoint framework does not rely on particular properties of these three adjoint pairs, it seems convenient to allow for a generalization of the implementation to, at least, a family of continuous t-norms, for any continuous t-norm can be interpreted as the ordinal sum of product and Łukasiewicz t-norms.

In order to maintain the most of the proposed implementation, it makes sense to consider an extra process in the homogenization process in order to further translate a program with new types of t-norms into one on which the original neural approach is still applicable.

Recall the definition of ordinal sum of a family of t-norms:

Definition 6 *Let $(\&_i)_{i\in\Lambda}$ be a family of t-norms and a family of non-empty pairwise disjoint subintervals $[a_i, b_i]$ of $[0,1]$. The* ordinal sum *of the summands $(a_i, b_i, \&_i)$, $i \in \Lambda$ is the t-norm $\&$ defined as*

$$\&(x,y) = \begin{cases} a_i + (b_i - a_i)\,\&_i(\frac{x-a_i}{b_i-a_i}, \frac{y-a_i}{b_i-a_i}) & x, y \in [a_i, b_i] \\ \min(x,y) & \textit{otherwise} \end{cases}$$

In order to simplify the notation of ordinal sum, let us introduce suitable functions for change of scale, to be able to switch between the intervals $[0,1]$ and $[a_i, b_i]$. Given the unit interval $[0,1]$ and a subset $[a_i, b_i] \subseteq [0,1]$, the function

$$f_i \colon [a_i, b_i] \to [0,1], \text{ with } f_i(x) = \frac{x - a_i}{b_i - a_i}$$

is bijective, and its inverse is

$$f_i^{-1} \colon [0,1] \to [a_i, b_i], \text{ with } f_i^{-1}(x) = a_i + (b_i - a_i)x$$

Now, given a t-norm $\&_i$ consider the following (adapted) t-norm[2]:

$$x \,\&_i^* \, y = \begin{cases} f_i^{-1}(f_i(x) \,\&_i\, f_i(y)) & \text{if } x, y \in [a_i, b_i] \\ \min\{x, y\} & \text{otherwise} \end{cases} \tag{1}$$

for all $x, y \in [0, 1]$.

With the notations above it is obvious that the ordinal sum of the summands $(a_i, b_i, \&_i)$, $i \in \Lambda$ can be written as

$$x \,\& \, y = \min\{x \,\&_i^* \, y \mid i \in \Lambda\} \tag{2}$$

With this expression for the ordinal sum, we can introduce a third type of transformation, to be applied to those rules of a homogeneous program with a *finite* ordinal sum in the body. i.e. $\Lambda = \{1, \ldots, n\}$.

*T*3. The homogeneous rule[3] $\langle A \leftarrow_s B_1 \,\&_s\, B_2, \vartheta \rangle$, where $\&$ is expressed as in (2) is substituted by the following $n + 1$ formulas:

$$\begin{gathered} \langle A_1 \leftarrow_s^* B_1 \,\&_1^*\, B_2, \vartheta \rangle \\ \vdots \\ \langle A_n \leftarrow_n^* B_1 \,\&_n^*\, B_2, \vartheta \rangle \\ \langle A \leftarrow_G \&_G(A_1, \ldots, A_n), \vartheta \rangle \end{gathered}$$

where $A_1, \ldots, A_n$ are fresh propositional symbols, and $\leftarrow_i^*$ are the adjoint implications of $\&_i^*$.

After applying this transformation to a homogeneous program, we obtain another homogeneous program in whose bodies no t-norm appears as an ordinal sum. This kind of homogenous program is called *sum-free homogeneous program* (in short *sf-homogeneous program*).

Example 1 *Consider the elements* $a_1 = 0.1$, $b_1 = 0.5$, $a_2 = 0.7$, $b_2 = 0.9$ *and the ordinal sum* $\&_s$ *defined as*

$$x \,\&_s \, y \begin{cases} f_1^{-1}(f_1(x) \,\&_P\, f_1(y)) & \textit{if } x, y \in [a_1, b_1] \\ f_2^{-1}(f_2(x) \,\&_L\, f_2(y)) & \textit{if } x, y \in [a_2, b_2] \\ \min\{x, y\} & \textit{otherwise} \end{cases}$$

Then if we have the homogeneous rule $\langle A \leftarrow_s B_1 \,\&_s\, B_2, 1 \rangle$*, applying T3, this is transformed in*[4]

$$\begin{aligned} \langle A_1 \leftarrow_P^* B_1 \,\&_P^*\, B_2, 1 \rangle &\quad \textit{sf-homogeneous} \\ \langle A_n \leftarrow_L^* B_1 \,\&_L^*\, B_2, 1 \rangle &\quad \textit{sf-homogeneous} \\ \langle A \leftarrow_G \&_G(A_1, \ldots, A_n), 1 \rangle &\quad \textit{sf-homogeneous} \end{aligned}$$

[2]For example, if we have that $[a_i, b_i] = [0, 1]$ then $x \,\&_P^* \, y = x \,\&_P \, y$.

[3]To simplify the presentation, let us assume that the body has just two arguments.

[4]Note that it is not necessary to specify the type of the implications in the rules because the weights are 1. Usually, we will omit the subscript in these cases.

The procedure to transform the rules of a program so that all the resulting rules are sf-homogeneous, is presented in Fig. 2. In its description by abuse of notation we use the terms T1-rule (resp. T2-rule, T3-rule) to mean an adequate input rule for transformation T1 (resp. T2, T3).

```
Program sf-Homogenization
  begin
    repeat
      for each T1-rule do
          Apply transformation T1
      end-for
      for each T2-rule do
          Apply transformation T2
      end-for
      for each T3-rule do
          Apply transformation T3
      end-for
    until neither T1- nor T2- nor T3-rules exist
  end
```

Figure 2: Pseudo-code for translating into a sf-homogeneous program.

3.1 Preservation of the semantics

It is necessary to check that the semantics of the initial program has not been changed by the transformation. The following results will show that every model of the sf-homogenized program $\mathbb{P}^*$ is also a model of the original program $\mathbb{P}$ and, in addition, the minimal model of $\mathbb{P}^*$ is also the minimal model of $\mathbb{P}$.

Theorem 1 *Let $\mathbb{P}$ be a homogeneous program, then every model of the program $\mathbb{P}^*$, obtained after to apply the sf-homogenization process to $\mathbb{P}$, is also a model of $\mathbb{P}$ when restricted to the variables occurring in $\mathbb{P}$.*

Proof: It will be sufficient to show that the transformation T3 satisfies that every model of its output is also a model of its input, as the proof for T1 and T2 was given in [6].

Assume that I is a model of the rules

$$\langle A_1 \leftarrow B_1 \,\&_1^*\, B_2, 1\rangle, \quad \ldots, \quad \langle A_n \leftarrow B_1 \,\&_n^*\, B_2, 1\rangle$$
$$\langle A \leftarrow_G \&_G(A_1, \ldots, A_n), 1\rangle$$

therefore we have

$$\hat{I}(B_1 \,\&_i^*\, B_2) \leq I(A_i) \quad \text{and} \quad \hat{I}(\&_G(A_1, \ldots, A_n)) \leq I(A)$$

for all $i \in \{1, \ldots, n\}$. Now, by monotonicity, we have

$$\&_G(\hat{I}(B_1 \&_1^* B_2), \ldots, \hat{I}(B_1 \&_n^* B_2)) \leq I(A)$$

Recall that we want to prove that I satisfies the rule $\langle A \leftarrow_i B_1 \,\&\, B_2, 1\rangle$, that is, $\hat{I}(B_1 \,\&\, B_2) \leq I(A)$, where $\&$ is an ordinal sum of $\&_i^*$, $i \in \{1, \ldots, n\}$. But this is true by Equation (2) above, since we have

$$\begin{array}{rcl} I(B_1) \,\&\, I(B_2) = & \min_{i \in \{1,\ldots,n\}}\{I(B_1) \&_i^* I(B_2)\} \\ = & \&_G(\hat{I}(B_1 \&_1^* B_2), \ldots, \hat{I}(B_1 \&_n^* B_2)) \\ \leq & I(A) \end{array}$$

QED

Theorem 2 *Given a program $\mathbb{P}$, the minimal model of the program $\mathbb{P}^*$ obtained after applying sf-homogenization, is also a model of $\mathbb{P}$ when restricted to variables in $\mathbb{P}$.*

The idea underlying the proof is to consider any model I of $\mathbb{P}$, then extend it to $\mathbb{P}^*$ in such a way that it is also a model of $\mathbb{P}^*$, finally use minimality on $\mathbb{P}^*$. The key point is to notice that, for every "fresh" propositional variable A_i introduced by the process, there is only one rule headed with A_i in the resulting program. This feature allows the extension of any model I to these new symbols in purely recursive terms.

Proof:

Let M^* be the minimal model of $\mathbb{P}^*$, and let M denote its restriction to $\mathbb{P}$. By the Theorem 1 we have that M is also a model of $\mathbb{P}$, so we have only to prove that it is minimal.

Once again, only the behaviour of transformation T3 has to be taken into account.

Given a model I of $\mathbb{P}$, consider a rule $\langle A_i \leftarrow B_1 \&_i^* B_2, 1\rangle$, where A_i is a propositional variable in $\mathbb{P}^*$ but not in $\mathbb{P}$. We argued above that there can be only one such rule headed with A_i, therefore the following extension of I makes sense:

$$I^*(A_i) = \hat{I}(B_1 \&_i^* B_2)$$

Obviously, by definition this extension I^* is also a model of $\mathbb{P}^*$, therefore the minimal model M^* of $\mathbb{P}^*$ satisfies $M^* \sqsubseteq I^*$. Now, by restricting the domain back to the variables in $\mathbb{P}$ we obtain $M \sqsubseteq I$. Therefore, M is the minimal model of $\mathbb{P}$.

QED

3.2 Complexity issues

In [6] it was shown that the complexity of the algorithm for transforming a multi-adjoint program into a homogeneous one is linear on the size of the program. Specifically, the following theorem was stated and proved

Theorem 3 *Let $\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_l), \vartheta\rangle$ be a rule with n connectives in the body ($n \geq 1$), then:*

- *The number of homogeneous rules obtained after applying the procedure is n, if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and $n+1$ otherwise.*
- *The number of transformations applied by the procedure is $n-1$ if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and n otherwise.*

Back to the sf-homogenization process, it can also be shown to be linear on the size of the program. The following theorem shows a precise calculation of the complexity of the homogenization procedure.

Theorem 4 *Let $\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_l), \vartheta\rangle$ be a rule with n connectives in the body ($n \geq 1$), out of which exactly m are constructed as ordinal sums of k_i basic connectives for each $i \in \{1, \ldots, m\}$ then:*

- *The number of homogeneous rules obtained after applying the procedure is bounded by $n + m + \sum_{i=1}^{m} k_i$, if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and $n + m + \sum_{i=1}^{m} k_i + 1$ otherwise.*
- *The number of transformations T_i applied by the procedure is $n + 2m - 1$ if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and $n + 2m$ otherwise.*

Proof: Concatenate the thesis of Theorem 3 with the fact that an application of T3 generates exactly $k+1$ rules, provided that the ordinal sum was built out of k intervals, because in each homogeneous rule there is only one operator in the body. QED

4 The neural implementation to use ordinal sums

A neural-based implementation of multi-adjoint logic program has already been presented, which is able to calculate the minimal model of a multi-adjoint logic program built solely from product, Gödel and Łukasiewicz adjoint pairs and weighted sums. In this section, we introduce an extension which is able to work on programs whose conjunctors are built as finite ordinal sums.

4.1 The structure of the neural net

A neural net is built from a sf-homogeneous program where each process unit is associated either to a propositional symbol of the initial program or to an sf-homogeneous rule. The state of the i-th neuron in the instant t is expressed by its output function, denoted $S_i(t)$. The state of the network can be expressed by means of a state vector $\vec{S}(t)$, whose components are the output of the neurons forming the network; the initial state of $\vec{S}$ is 0 for all the components except those representing a propositional variable, say A, in which case its value is defined as:

$$S_A(0) = \begin{cases} \vartheta_A & \text{if } \langle A \leftarrow 1, \vartheta_A\rangle \in \mathbb{P}, \\ 0 & \text{otherwise.} \end{cases}$$

where $S_A(0)$ denotes the component associated to a propositional symbol A.

The connection between neurons is denoted by a matrix of weights W, in which w_{kj} indicates the existence or absence of connection between unit k and unit j; if the neuron represents a weighted sum, then the matrix of weights also represents the weights associated to any of the inputs. The weights of the connections related to neuron i (that is, the i-th row of the matrix W) are represented by an n-upla $W_{i\bullet}$, and are allocated in an internal vector register of the neuron, which can be seen as a distributed information system.

The initial truth-value of the propositional symbol or sf-homogeneous rule v_i is loaded in the internal register, together with a signal m_i to distinguish whether the neuron is associated either to a fact or to a rule; in the latter case, information about the type of operator is also included. Moreover, it is necessary introduce the (sub-)intervals on which the new connective will be operating, i.e. to set the values of the extremes of interval a_i and b_i. Therefore, there are four vectors: the first storing the truth-values $\vec{v}$ of atoms and sf-homogeneous rules, the second $\vec{m}$ storing the type of the neurons in the net and the last two representing the (sub-)intervals of the components of the ordinal sums.

A signal m_i indicates the functioning mode of the neuron. If $m_i = 1$, then the neuron is assumed to be associated to a propositional symbol, and its next state is the maximum value among all the operators involved in its input, its previous state, and the initial truth-value v_i. More precisely:

$$S_i(t+1) = \max\left\{ v_i, \max_k\{S_k(t) \mid w_{ik} > 0\} \right\}$$

Note that for this case, the values introduced as extremes of interval are, naturally, $a_i = 0$ and $b_i = 1$.

When a neuron is associated to the product, Gödel, or Łukasiewicz implication, respectively, then the signal m_i is set to 2, 3, and 4, respectively. Its input is formed by the external value v_i of the rule, the outputs of the neurons associated to the body of the implication and the extremes a_i, b_i of the interval of each connective.

The output of the neuron mimics the behaviour of the implication in terms of the adjoint property when a rule of type m_i has been used.

Before specifying the output of each neuron i in the net, we need to consider the following set, which takes care of the truth-value assigned to the neuron and its input values. We define the following set of relevant values for the neuron i:

$$J_i(t) = \{v_i\} \cup \{S_k(t) \mid w_{ik} > 0\}$$

Now we can specify the output in the next instant of each neuron:

$$S_i(t+1) = \begin{cases} \&_p^*(J_i(t)) & \text{if } m_i = 2 \\ \&_G(J_i(t)) & \text{if } m_i = 3 \\ \&_L^*(J_i(t)) & \text{if } m_i = 4 \end{cases}$$

Note that we do not use the starred version for the Gödel connective, since the default connective in the definition of ordinal sums is precisely the minimum.

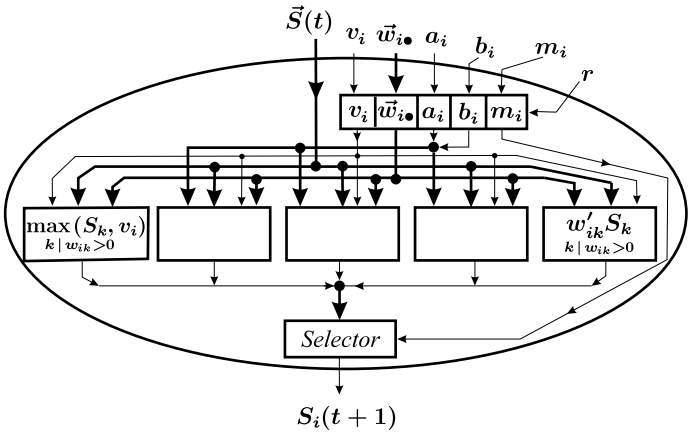

Figure 3: A generic neuron.

When $m_i = 5$ the value of the registers a_i and b_i is irrelevant. In this case the neuron operates as a weighted sum which outputs

$$S_i(t+1) = \sum_{k \mid w_{ik}>0} w'_{ik} S_k(t) \quad \text{where} \quad w'_{ik} = \frac{w_{ik}}{\sum_{k \mid w_{ik}>0} w_{ik}}$$

A generic neuron is shown in Fig. 3 above.

Example 2 Consider the conjunctor $\&_s = \{(0.1, 0.5, \&_P), (0.7, 0.9, \&_L)\}$ defined as an ordinal sum and the following toy program

$$\langle p \leftarrow_s q \,\&_s\, r, 1\rangle \quad \langle q \leftarrow 1, 0.5\rangle \quad \langle r \leftarrow 1, 0.8\rangle$$

After sf-homogenization we obtain a program with the rules

$$\langle p1 \leftarrow q \,\&_1^*\, r, 1\rangle \quad \langle p2 \leftarrow q \,\&_2^*\, r, 1\rangle \quad \langle p \leftarrow_G p_1 \,\&_G\, p_2, 1\rangle$$

and the two initial facts

$$\langle q \leftarrow 1, 0.5\rangle \quad \langle r \leftarrow 1, 0.8\rangle$$

Thus, we have five propositional symbols, out of which two are fresh, and five rules. As a result, the neural network has six neurons, three associated to initial propositional symbols p, q, r and three to the proper sf-homogeneous rules.

The values of the first neuron associated to p are the following: $v_1 = 0$ because there is not a fact information about p, $m_1 = 1$ since p is a propositional symbol, $a_1 = 0$ and $b_1 = 1$ by the previous reason and its row in the matrix W is $W_{1\bullet} = (0, 0, 0, 0, 0, 1)$ because only the last rule has p in the head. The two following neurons are associated to two propositional symbols and then their values are calculated similarly.

The fourth neuron is associated to the rule $\langle p_1 \leftarrow q \,\&_1^* \, r, 1\rangle$ and then $v_4 = 1$ (it is the weight of the rule), $m_4 = 2$ by the product conjunctor, $a_4 = 0.1$ and $b_4 = 0.5$ because it comes from the first interval of the ordinal sum $\&_s$, and $W_{4\bullet} = (0, 1, 1, 0, 0, 0)$ since it uses the output of the second and third neurons which are associated to q and r respectively. Similarly, we can obtain the initial values for the rest of the neurons.

Finally, to initialize the net we introduce:

- The vector $\vec{v} = (0, 0.5, 0.8, 1, 1, 1)$.
- The vector $\vec{m} = (1, 1, 1, 2, 4, 3)$.
- The vector $\vec{a} = (0, 0, 0, 0.1, 0.7, 0)$.
- The vector $\vec{b} = (1, 1, 1, 0.5, 0.9, 1)$.
- The matrix

$$W = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & \cdot \end{pmatrix}$$

4.2 Implementation of the neural network

A number of simulations have been obtained through a MATLAB implementation in a conventional sequential computer. A high level description of the implementation is given below:

1. **Initialize** the network is with the appropriate values of $\vec{v}$, $\vec{m}$, $\vec{a}$, $\vec{b}$, W and, in addition, a tolerance value *tol* to be used as a stop criterion. The output $S_i(t)$ of the neurons associated to facts (which are propositional variables, so $m_i = 1$) are initialized with its truth-value v_i.
2. **Repeat** Update all the states S_i of the neurons of the network :
   (a) If $m_i = 1$, then:
      i. Construct the following set, which amounts to collect all the rules with head A:
      $$J_i(t) = \{v_i\} \cup \{S_k(t) \mid w_{ik} > 0\}$$
      ii. Then, update the state of neuron i as follows:
      $$S_i(t+1) = \max J_i(t)$$
   (b) If $m_i = 2, 3$, then:
      i. Find the outputs of the neurons j (if any) which operate on the neuron i and consider v_i, that is, construct the set
      $$J_i(t) = \{v_i\} \cup \{S_k(t) \mid w_{ik} > 0\}$$

ii. Then, update the state of neuron i as follows:

$$S_i(t+1) = \begin{cases} \&_p^*(J_i(t)) & \text{if } m_i = 2 \\ \&_G(J_i(t)) & \text{if } m_i = 3 \\ \&_L^*(J_i(t)) & \text{if } m_i = 4 \end{cases}$$

(c) If $m_i = 5$, then the neuron corresponds to an aggregator, and its update follows a different pattern:

i. Determine the set $K_i = \{j \mid w_{ij} > 0\}$ and calculate $sum = \sum_{K_i} w_{ij}$

ii. Update the neuron as follows:

$$S_i(t+1) = \frac{1}{sum} \sum_{K_i} w_{ij} \cdot S_j(t)$$

Until the stop criterion $\|\vec{S}(t) - \vec{S}(t-1)\|_2 < tol$ is fulfilled, where $\|\cdot\|_2$ denotes euclidean distance.

Example 3 *Consider a program with facts* $\langle p \leftarrow 1, 0.3\rangle$ *and* $\langle q \leftarrow 1, 0.4\rangle$ *and rules* $\langle p \leftarrow_s s \,\&_s\, q, 1\rangle, \langle s \leftarrow_s q, 0.7\rangle, \langle p \leftarrow_L s, 0.5\rangle$ *where* $\&_s = \{(0.1, 0.5, \&_P), (0.7, 0.9, \&_L)\}$ *is an ordinal sum.*

Then, the sf-homogeneous program is

$$\begin{array}{ccc} \langle p_1 \leftarrow s \,\&_1^*\, q, 1\rangle & \langle p_2 \leftarrow s \,\&_2^*\, q, 1\rangle & \langle p \leftarrow p_1 \,\&_G\, p_2, 1\rangle \\ \langle s_1 \leftarrow_1^* q, 0.7\rangle & \langle s_2 \leftarrow_2^* q, 0.7\rangle & \langle s \leftarrow s_1 \,\&_G\, s_2, 1\rangle \\ \langle p \leftarrow_L s, 0.5\rangle & & \end{array}$$

with the same facts $\langle p \leftarrow 1, 0.3\rangle$ and $\langle q \leftarrow 1, 0.4\rangle$.

The net for the program is sketched in Fig. 4, and consists of ten neurons: three of which represent variables p, q, s, and the rest are needed to represent the rules (three for each ordinal sum rule and another one for the Łukasiewicz rule). The initial values of the net are:

- The vector $\vec{v} = (0.3, 0.4, 0, 1, 1, 1, 0.7, 0.7, 1, 0.5)$.
- The vector $\vec{m} = (1, 1, 1, 2, 4, 3, 2, 4, 3, 4)$.
- The vector $\vec{a} = (0, 0, 0, 0.1, 0.7, 0, 0.1, 0.7, 0, 0)$.
- The vector $\vec{b} = (1, 1, 1, 0.5, 0.9, 1, 0.5, 0.9, 1, 1)$.

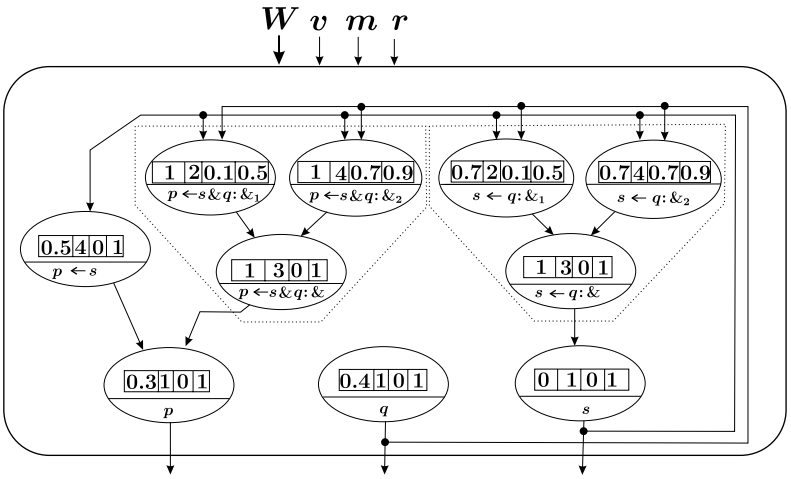

Figure 4: A network for Example 3.

- The matrix

$$W = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

After running the net, its state vector gets stabilized at

$$\vec{S} = (0.325, 0.4, 0.4, 0.325, 0.4, 0.325, 0.4, 0.4, 0.4, 0.32)$$

where the last seven components correspond to hidden neurons, the first ones are interpreted as the obtained truth-value for p, q and s.

4.3 Relating the net and $T_{\mathbb{P}}$

In this section we relate the behavior of the components of the state vector with the immediate consequence operator. To begin with, it is convenient to recall that the functions S_i implemented by each neuron are non-decresing. The proof is straightforward, after analysing the different cases arising from the type of neuron (i.e. the register m_i).

Regarding the soundness of the implementation sketched above, the following theorem can be obtained.

Theorem 5 *Given a sf-homogeneous program $\mathbb{P}$ and a symbol A, then*

$$T_{\mathbb{P}}{}^n(\triangle)(A) = S_A(2n-2) \quad \textit{for} \quad n \geq 1.$$

The proof of this theorem is similar to that given in [6], thus we do not repeat it here. By the previous theorem we obtain the following corollary:

Corollary 1 *Given a homogeneous program $\mathbb{P}$ and a propositional symbol A, then the sequence $S_A(n)$ approximates the value of the least model of $\mathbb{P}$ in A up to any prescribed level of precision.*

Regarding the convergence of the network, by Cor. 1 and Knaster-Tarski theorem (assuming continuity of $T_{\mathbb{P}}$) it is the case that the net always obtains an approximation to the fixed point up to any level of precision. In the case that $T_{\mathbb{P}}$ reaches the fixpoint after a finite number of steps, then the network always converges to the *exact* value of the minimal model. This happens for some special types of programs, for instance in [9] termination in finitely many steps is proved for homogeneous programs with only one conjunction connective and without aggregators.

5 Related approaches

The use of neural networks in the context of logic programming is not a new idea; for instance, Eklund and Klawonn showed in [2] how fuzzy logic programs can be transformed into neural nets, where adaptations of uncertainties in the knowledge base increase the reliability of the program and are carried out automatically. On the other hand, Hölldobler and Kalinke implemented in [3] the fixpoint of the $T_{\mathbb{P}}$ operator for a certain class of classical propositional logic programs (called acyclic logic programs) by using a 3-layered recurrent neural network; this result is later extended in [4] to deal with the first order case.

Our approach somehow tries to join the two approaches above, and it is interesting since our logic is much richer than classical or the usual versions of fuzzy logic in the literature, although we only consider the propositional case. Although there are some results regarding the expressive power of feed-forward multilayered neural nets, such as Kůrková's theorem [5], the structure of our net is not described as an n-layered network, instead a more straightforward approach is used.

The authors presented in [7] a different neural approach which allows ordinal sums in the initial program as well. In that paper, the homogenization process introduced in [6] was assumed to be applied. In this paper, we extend, in a natural way, the neural net in [6] making some changes due to the introduction of the new rule T3 for the sf-homogenization.

Specifically, in the net for ordinal sums presented in [7], each neuron associated to an ordinal sum needs as input, apart from the outputs of the neurons in charge of the intermediate calculations, the outputs of the neurons associated to the propositional symbols occurring in the body of the rule. This resulted in a poor performance due to, among other reasons, to the need of using output obtained two

steps back, i.e. the calculation of $S_i(t+1)$ depend on some $S_k(t-1)$. The solution to this problem, also given in [7], introduced as a side effect that the output of the net could not be related to the $T_{\mathbb{P}}$ operator in the same easy manner as in [6].

In the approach given in this paper, we solve *simultaneously* the two technical problems stated in the previous paragraph, in that only the outputs obtained in the previous time step are needed and, moreover, the statement of the theorem relating the net and the $T_{\mathbb{P}}$ operator can be recovered (Theorem 5). This is a beneficial effect of the modification proposed for the homogenization process, i.e. use sf-homogenization instead of the homogenization of [6], which has just been introduced in this paper.

6 Conclusions and future work

It has been shown that the homogenization process for multi-adjoint logic programs can be adapted so that it can "decompose" conjunctors defined as an ordinal sum. From the theoretical point of view, the neural implementation proposed here, inherits the intrinsic relationship between the neural net and the $T_{\mathbb{P}}$ operator; whereas, from the practical point of view, this approach is potentially as well-behaved as that presented in [7].

A different approach to the solution of the problem studied in this paper would be to modify directly the neural implementation so that the ordinal sum constructor can be conveniently represented by the net. Then, it would be interesting to make a comparison of the efficiency of the treatment of compound conjunctors via this extended sf-homogenization an the modified neural approach. As future work, we are planning a thorough comparison between the different approaches.

References

[1] C.V. Damásio and L. Moniz Pereira. Monotonic and residuated logic programs. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU'01*, pages 748–759. Lect. Notes in Artificial Intelligence, 2143, 2001.

[2] P. Eklund and F. Klawonn. Neural fuzzy logic programming. *IEEE Tr. on Neural Networks*, 3(5):815–818, 1992.

[3] S. Hölldobler and Y. Kalinke. Towards a new massively parallel computational model for logic programming. In *ECAI'94 workshop on Combining Symbolic and Connectioninst Processing*, pages 68–77, 1994.

[4] S. Hölldobler, Y. Kalinke, and H.-P. Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11(1):45–58, 1999.

[5] V. Kůrková. Kolmogorov's theorem and multilayer neural networks, *Neural Networks* 5: 501-506, 1992.

[6] J. Medina, E. Mérida-Casermeiro, and M. Ojeda-Aciego. A neural implementation of multi-adjoint logic programming. *Journal of Applied Logic*, 2(3):301-324, 2004.

[7] J. Medina, E. Mérida-Casermeiro, and M. Ojeda-Aciego. Decomposing Ordinal Sums in Neural Multi-Adjoint Logic Programs. *Lect. Notes in Artificial Intelligence* 3315:717–726, 2004.

[8] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. In *Lect. Notes in Artificial Intelligence* 2173:351–364, 2001.

[9] L. Paulík. Best Possible Answer is Computable for Fuzzy SLD-Resolution In *Proceedings of Gödel'96*: Logical Foundations of Mathematics, Computer Science and Physics; Kurt Gödel's Legacy, pages 257–266, 1997.

[10] M. H. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[11] P. Vojtáš. Fuzzy logic programming. *Fuzzy Sets and Systems*, 124(3):361–370, 2001.