Decomposing Ordinal Sums in Neural Multi-Adjoint Logic Programs

J. Medina, E. Mérida-Casermeiro, and M. Ojeda-Aciego

Dept. Matemática Aplicada. Universidad de Málaga, Email:
{jmedina,merida,aciego}@ctima.uma.es

Abstract. The theory of multi-adjoint logic programs has been introduced as a unifying framework to deal with uncertainty, imprecise data or incomplete information. From the applicative part, a neural net based implementation of homogeneous propositional multi-adjoint logic programming on the unit interval has been presented elsewhere, but restricted to the case in which the only connectives involved in the program were the usual product, Gödel and Łukasiewicz together with weighted sums.
A modification of the neural implementation is presented here in order to deal with a more general family of adjoint pairs, including conjunctors constructed as an ordinal sum of a finite family of basic conjunctors. This enhancement greatly expands the scope of the initial approach, since every t-norm (the type of conjunctor generally used in applications) can be expressed as an ordinal sum of product, Gödel and Łukasiewicz conjunctors.

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 inspired by the structure of human brain, adaptation capabilities, and fault tolerance.

Using neural networks in the context of logic programming is not a completely novel idea; for instance, in [2] it is shown 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.

Regarding the approximation of the semantics of logic programs, the fixpoint of the $T_{\mathbb{P}}$ operator for a certain class of classical propositional logic programs (called acyclic logic programs) is constructed in [3] by using a 3-layered recurrent neural network, as a means of providing a massively parallel computational model for logic programming; this result is later extended in [4] to deal with the first order case.

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. This neural-like implementation of multi-adjoint logic programming was presented with the restriction that the only

connectives involved in the program were the usual product, Gödel and Łukasiewicz together with weighted sums. Since 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 purpose of this paper is to present a refined version of the neural implementation, such that conjunctors which are built as ordinal sums of product, Gödel and Łukasiewicz t-norms are decomposed into its components and, as a result, the original neural approach is still applicable. It is worth to remark that the learning capabilities of neural networks are not used in the implementation; this should not be considered a negative feature, because it is precisely in the final goal of this research line, a neural approach to abductive multi-adjoint logic programming, when learning will play a crucial role.

The structure of the paper is as follows: In Section 2, the syntax and semantics of multi-adjoint logic programs are introduced; in Section 3, the new proposed neural model for homogeneous multi-adjoint programs is presented in order to cope with conjunctors defined as ordinal sums, a high level implementation is introduced and proven to be sound. 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 [7].

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,*
   (c) *For any* $x, y, z \in P$*, we have* $x \preceq (y \leftarrow_i z)$ *if and only if* $(x \,\&_i\, z) \preceq y$*.*

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 and semantics

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

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 body1 and a *query* (or *goal*) is a propositional symbol intended as a question $?A$ prompting the system.

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 $\langle [0, 1], \leq \rangle$.*

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.*
3. *An element $\lambda \in L$ is a* correct answer *for a program $\mathbb{P}$ and a query $?A$ if for any interpretation $I \in \mathcal{I}_{\mathfrak{L}}$ which is a model of $\mathbb{P}$ we have $\lambda \leq I(A)$.*

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 [9], which can be generalised to the multi-adjoint framework 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 [7], then the least model can be reached in at most countably many iterations beginning with the least interpretation, that is, the least model is $T_{\mathbb{P}} \uparrow \omega(\triangle)$.

2.2 Homogeneous programs

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 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, $(\&_i, \leftarrow_i)$ is an adjoint pair, and $@$ is an aggregator.

The homogeneous rules represent exactly the simplest type of (proper) rules one can have in a program. The way in which the translation of a general multi-adjoint program is homogenized is irrelevant for the purposes of this paper; anyway, it is worth mentioning that it is a model preserving procedure with linear complexity.

For the sake of self-contention, a brief overview of the neural network from [6] is presented below.

2.3 The restricted neural implementation

A neural network was considered in which each process unit is associated either to a propositional symbol of the initial program or to an homogeneous rule of the transformed program. 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 $\boldsymbol{S}(t)$, whose components are the output of the neurons forming the network; the initial state of $\boldsymbol{S}$ is 0 for all the components except those representing a propositional variable, say A, in which case its value is defined to be:

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

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 a vector $\boldsymbol{w}_{i\bullet}$, 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 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. Therefore, there are two vectors: one storing the truth-values $\boldsymbol{v}$ of atoms and homogeneous rules, and another $\boldsymbol{m}$ storing the type of the neurons in the net.

The 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:

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

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, and the outputs of the neurons associated to the body of the implication.

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; specifically, the output in the next instant will be:

$$S_i(t+1) = \begin{cases} v_i \prod\limits_{k \mid w_{ik}>0} S_k(t) & \text{if } m_i = 2 \\ \min\left\{ v_i, \min\limits_k \{S_k(t) \mid w_{ik} > 0\} \right\} & \text{if } m_i = 3 \\ \max\{ v_i + \sum\limits_{k \mid w_{ik}>0} (S_k(t) - 1), 0\} & \text{if } m_i = 4 \end{cases}$$

A neuron associated to a weighted sum has signal $m_i = 5$, and its output is

$$S_i(t+1) = \sum_k w'_{ik} S_k(t) \qquad \text{where} \qquad w'_{ik} = \frac{w_{ik}}{\sum\limits_r w_{ir}}$$

3 Towards a new model of generic neuron

As stated above, the neural net implementation of the immediate consequences operator of an homogeneous program was introduced 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, since any continuous t-norm can be interpreted as the ordinal sum of product and Łukasiewicz t-norms.

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

Definition 6. *Let $(\&_i)_{i\in A}$ 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 $(a_i, b_i, \&_i)$, $i \in A$ is the t-norm $\&$ defined as*

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

3.1 The proposed model of generic neuron

The model of neuron presented in [6] has to be modified in order to be able to represent conjunctors defined as ordinal sums of product and Łukasiewicz conjunctions.

To begin with, two new registers are introduced so that it is possible to represent the (sub-)interval on which the connective will be operating, i.e. to set the values of the points a_i and b_i.

The generic neuron shown in Fig. 1 has a binary signal r, which is shared by all the neurons. If $r = 1$, then it is possible to modify the content of the internal registers v, $\boldsymbol{w}_{i.}$, m_i, a_i and b_i by means of the external signals of the same name. If $r = 0$,

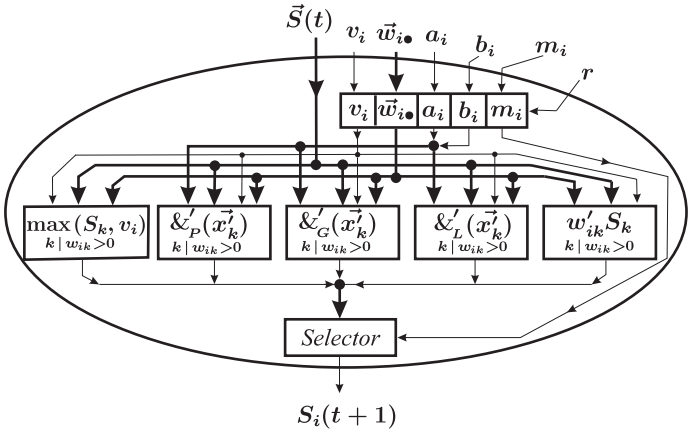

Fig. 1. The proposed generic neuron.

then neuron computes an output $S_i(t+1)$ in terms of its own internal registers and its network inputs (that is, the outputs of the rest of neurons in the previous step $\boldsymbol{S}(t)$).

Depending on the content of the register m_i, the selector generates the component of the generic neuron which gets activated. When $m_i = 1$ the value of the registers a_i and b_i is irrelevant; this is also true when $m_i = 5$. In this case the neuron operates as a weighted sum which outputs

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

In the cases $m_i = 2$, $m_i = 3$ and $m_i = 4$ the neuron uses the input $\boldsymbol{S}(t)$ and the value v_i to compute a vector $\boldsymbol{x}_i{}'$ as follows:

- Only the value v_i and the components k of $\boldsymbol{S}(t)$ such that $w_{ik} > 0$ will be considered.
- The vector $\boldsymbol{x}'$ is computed as $x'_k = \begin{cases} 1 & S_k(t) \geq b_i \\ \frac{S_k(t)-a_i}{b_i-a_i} & a_i \leq S_k(t) < b_i \\ 0 & S_k(t) < a_i \end{cases}$
- The output of the neuron is[1] $S_i(t+1) = \begin{cases} a_i + (b_i - a_i)\&_P(\boldsymbol{x}') & \text{if } m_i = 2 \\ a_i + (b_i - a_i)\&_G(\boldsymbol{x}') & \text{if } m_i = 3 \\ a_i + (b_i - a_i)\&_L(\boldsymbol{x}') & \text{if } m_i = 4 \end{cases}$

3.2 Neural representation of ordinal sums

Consider a conjunctor represented as an ordinal sum

$$\& = \{\langle a_1, b_1, \&_1\rangle, \langle a_2, b_2, \&_2\rangle, \ldots, \langle a_k, b_k, \&_k\rangle\}$$

[1] The functions corresponding to each case are represented in Fig. 1 as $\&'_P$, $\&'_G$, $\&'_L$, respectively.

where $\&_i$ can be either $\&_P$ or $\&_L$ (since Gödel conjunctor operates by default out of the intervals $[a_i, b_i)$).

The implementation of such a compound connective needs $k+1$ neurons, the first k are associated to each subinterval of the ordinal sum, and the final one collects all their outputs and generates the final output of the sum. As shown in Fig. 2, all the neurons of the block associated with the ordinal sum $\&$ receive the output of the neurons associated to the propositions in the body of the rule, whereas the collecting neuron receives the output of the rest of the neurons integrating the block of the ordinal sum.

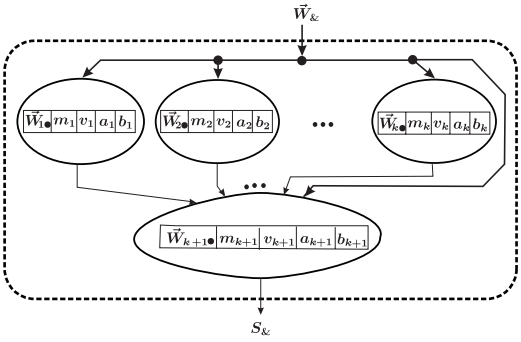

Fig. 2. Neural version of a general ordinal sum.

Let us see the process working on a specific toy example:

Example 1. Consider the ordinal sum defined by $\& = \{(0.1, 0.5, \&_P), (0.7, 0.9, \&_L)\}$ and the program with one rule and two facts: $\langle p \leftarrow_{\&} q \,\&\, s, 0.6\rangle, \langle q, 0.4\rangle, \langle s, 0.3\rangle$. The neural representation needs 6 neurons: three are associated to the propositions p, q and s, and the other three are used to implement the rule.

The initialization of the registers allocates the values for the vectors $\boldsymbol{m}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{v}$ and for the weights matrix as shown below

$$\begin{aligned} \boldsymbol{m} &= (1, 1, 1, 2, 4, 3) \\ \boldsymbol{a} &= (0.0, 0.0, 0.0, 0.1, 0.7, 0.0) \\ \boldsymbol{b} &= (1.0, 1.0, 1.0, 0.5, 0.9, 1.0) \\ \boldsymbol{v} &= (0.0, 0.4, 0.3, 0.6, 0.6, 0.6) \end{aligned} \qquad \boldsymbol{W} = \begin{pmatrix} 0\,0\,0\,0\,0\,1 \\ 0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,0\,0\,0 \\ 0\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,0 \end{pmatrix}$$

3.3 Implementation

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 with the appropriate values of $\boldsymbol{v}$, $\boldsymbol{m}$, 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 set $J_i = \{j \mid w_{ij} = 1\}$. In this case, this amounts to collect all the rules with head A.
      ii. Then, update the state of neuron i as follows:

$$S_i(t) = \begin{cases} \max\{v_i, \max_{J_i} S_j(t-1)\} & \text{if } J_i \neq \varnothing \\ v_i & \text{otherwise} \end{cases}$$

   (b) If $m_i = 2, 3, 4$, then:
      i. Find the neurons j (if any) which operate on the neuron i, that is, construct the set $J_i = \{j \mid w_{ij} = 1\}$.
      ii. Then, update the state of neuron i as follows:

$$S_i(t) = \begin{cases} v_i \prod_{J_i} S_j(t-1) & \text{if } m_i = 2 \\ \min\{v_i, \min_{J_i} S_j(t-1)\} & \text{if } m_i = 3 \\ \max\{v_i + \sum_{J_i}(S_j(t-1) - 1), 0\} & \text{if } m_i = 4 \end{cases}$$

   Recall that when $m_i = 2, 3, 4$ the neuron corresponds to a product, Gödel, Łukasiewicz (resp.) implication.
   (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:

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

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

Example 2. Consider a program with facts $\langle p \leftarrow 1, 0.3\rangle$ and $\langle q \leftarrow 1, 0.4\rangle$ and three rules $\langle p \leftarrow_{\&} s \,\&\, q, 0.8\rangle, \langle s \leftarrow_{\&} q, 0.7\rangle, \langle p \leftarrow_{L} s, 0.5\rangle$ where the conjunction $\&$ is defined as an ordinal sum by $\& = \{(0.1, 0.5, \&_P), (0.7, 0.9, \&_L)\}$.

The net for the program will consist of ten neurons, and is sketched in Fig. 3, 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).

$$\begin{aligned} \boldsymbol{m} &= (1,1,1,2,4,3,2,4,3,4) \\ \boldsymbol{a} &= (0.0,0.0,0.0,0.1,0.7,0.0,0.1,0.7,0.0,0.0) \\ \boldsymbol{b} &= (1.0,1.0,1.0,0.5,0.9,1.0,0.5,0.9,1.0,1.0) \\ \boldsymbol{v} &= (0.3,0.4,0.0,0.8,0.8,0.8,0.7,0.7,0.7,0.5) \end{aligned} \qquad \boldsymbol{W} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

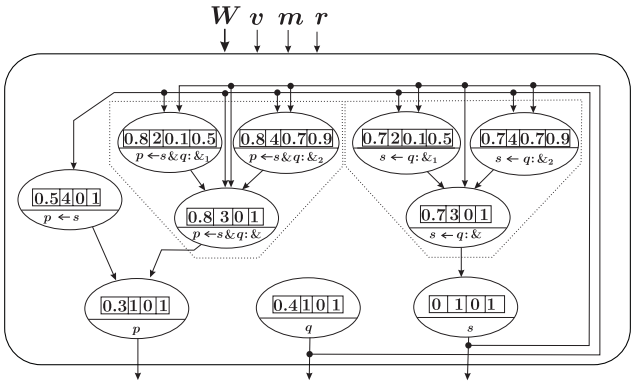

Fig. 3. A network for Example 2.

After running the net, its state vector gets stabilized at

$$\boldsymbol{S} = (0.325, 0.4, 0.4, 0.325, 0.7, 0.325, 0.4, 0.7, 0.4, 0)$$

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

3.4 Relating the net and $\boldsymbol{T}_{\mathbb{P}}$

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, although space restrictions do not allow to include the proof.

Theorem 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.*

4 Conclusions and future work

A new neural-like model has been proposed which extends that recently given to multi-adjoint logic programming. The model mimics the consequences operator in order to obtain the least fixpoint of the immediate consequences operator for a given multiadjoint logic program. This way, it is possible to obtain the computed truth-values of all propositional symbols involved in the program in a parallel way. This extended approach considers, in addition to the three most important adjoint pairs in the unit interval (product, Gödel, and Łukasiewicz) and weighted sums, the combinations as finite ordinal sums of the previous conjunctors.

We have decided to extend the original generic model of neuron, capable of adapting to perform different functions according with its inputs. Although it is possible to

consider simpler units, the price to pay is to consider a set of *different units* each type representing a different type of homogeneous rule or proposition. This way one has both advantages and disadvantages: the former are related to the easier description of the units, whereas the latter arise from the greater complexity of the resulting network. A complete analysis of the compromise between simplicity of the units and complexity of the network will be the subject of future work.

Another line of future research deals with further developing the neural approach to abductive multi-adjoint logic programming [5]; it is in this respect where the learning capabilities of neural networks are proving to be an important tool.

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. J. Medina, E. Mérida-Casermeiro, and M. Ojeda-Aciego. A neural approach to abductive multi-adjoint reasoning. In *AI - Methodologies, Systems, Applications. AIMSA'02*, pages 213–222, Lect. Notes in Computer Science 2443, 2002.
6. J. Medina, E. Mérida-Casermeiro, and M. Ojeda-Aciego. A neural approach to extended logic programs. In *7th Intl Work Conference on Artificial and Natural Neural Networks, IWANN'03*, pages 654–661. Lect. Notes in Computer Science 2686, 2003.
7. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. In *Logic Programming and Non-Monotonic Reasoning, LPNMR'01*, pages 351–364. Lect. Notes in Artificial Intelligence 2173, 2001.
8. 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.
9. 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.
10. P. Vojtáš. Fuzzy logic programming. *Fuzzy Sets and Systems*, 124(3):361–370, 2001.