11

Connectionist Representation of Multi-Valued Logic Programs

Ekaterina Komendantskaya, Máire Lane and Anthony Karel Seda

Department of Mathematics, University College Cork, Cork, Ireland
komendantskaya@gmail.com, maireln@bcri.ucc.ie, a.seda@ucc.ie[†]

Summary. Hölldobler and Kalinke showed how, given a propositional logic program P, a 3-layer feedforward artificial neural network may be constructed, using only binary threshold units, which can compute the familiar immediate-consequence operator T_P associated with P. In this chapter, essentially these results are established for a class of logic programs which can handle many-valued logics, constraints and uncertainty; these programs therefore represent a considerable extension of conventional propositional programs. The work of the chapter basically falls into two parts. In the first of these, the programs considered extend the syntax of conventional logic programs by allowing elements of quite general algebraic structures to be present in clause bodies. Such programs include many-valued logic programs, and semiring-based constraint logic programs. In the second part, the programs considered are bilattice-based annotated logic programs in which body literals are annotated by elements drawn from bilattices. These programs are well-suited to handling uncertainty. Appropriate semantic operators are defined for the programs considered in both parts of the chapter, and it is shown that one may construct artificial neural networks for computing these operators. In fact, in both cases only binary threshold units are used, but it simplifies the treatment conceptually to arrange them in so-called multiplication and addition units in the case of the programs of the first part.

11.1 Introduction

In their seminal paper [1], Hölldobler and Kalinke showed how, given a propositional logic program P, one may construct a 3-layer feedforward artificial neural network (ANN), having only binary threshold units, which can compute the familiar immediate-consequence operator T_P associated with P. This result has been taken as the starting point of a line of research which forms one component of the general problem of integrating the logic-based and connectionist or neural-network-based approaches to

[†] Author for correspondence: A.K. Seda.

computation. Of course, the objective of integrating these two computing paradigms is to combine the advantages to be gained from connectionism with those to be gained from symbolic computation. The papers [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1, 2, 13, 14, 15, 16, 17, 18, 19] represent a small sample of the literature on this topic. In particular, the papers just cited contain extensions and refinements of the basic work of Hölldobler et al. in a number of directions including inductive logic programming, modal-logic programming, and distributed knowledge representation. In the large, the problem of integrating or combining these two approaches to computation has many aspects and challenges, some of which are discussed in [3, 12].

Our objective and theme in this chapter is to present generalizations of the basic results of [1] to classes of extended logic programs P which can handle many-valued logics, constraints and uncertainty; in fact, the programs we discuss are annotated logic programs. Thus, we are concerned primarily with the computation by ANN of the semantic operators determined by such extended logic programs. Therefore, we begin with a brief discussion of these programs, their properties and powers of knowledge representation.

As is well-known, there exists a great variety of many-valued logics of interest in computation, see [20] for a survey. For example, Lukasiewicz and Kleene introduced several three-valued logics [20]. Then infinite-valued Lukasiewicz logic appeared, and there are various other many-valued logics of interest, such as fuzzy logic and intuitionistic logic. In another direction, Belnap studied bilattice-based logics for reasoning with uncertainties, and his work was further developed in [21, 22]. Most of these logics have been adapted to logic programming: for example, see [23] for logic programs interpreted by arbitrary sets, [24] for applications of Kleene's logic to logic programming, [25] for semiring-based constraint logic programming, [26, 27] for fuzzy logic programming, and [28, 29, 30] for (bi)lattice-based logic programming. See also [31] for a very general algebraic analysis of different many-valued logic programs. However, in the main, there have been three approaches to many-valued logic programming, as follows.

First, *annotation-free logic programs* were introduced by Fitting in [24] and further developed in [25, 28, 29, 32]. They are formally the same as two-valued logic programs in that the clauses of an annotation-free logic program are exactly the same as those for a two-valued logic program. But, whilst each atomic ground formula of a two-valued logic program is given an interpretation in $\{\text{true}, \text{false}\}$, an atomic annotated formula of an annotation-free logic program receives its interpretation in an arbitrary set carrying idempotent, commutative and associative operations which model logical connectives, as we illustrate in more detail in Section 11.3. Next, *implication-based logic programs* were introduced by Van Emden in [33] and were designed in order to obtain a simple and effective proof procedure. Van Emden considered the particular case of the set $[0, 1]$ of truth values, but this has since been extended, see for example [34]. Much of this work carries over without difficulty to an arbitrary set with idempotent, commutative and associative operations. Van

Emden used the conventional syntax for logic programs, except for having clauses of the form

$$A \leftarrow \boxed{f} B_1, \ldots, B_n,$$

where f is a *factor* or *threshold* taken from the interval $[0, 1]$ of real numbers. The atoms A, $B_1, \ldots, B_n$ receive their interpretation from the interval $[0, 1]$, and are such that the value of the head A of a clause has to be greater than or equal to $f \times \min(|B_1|, \ldots, |B_n|)$. Finally, *annotated (or signed) logic programs* require each atom in a given clause to be annotated (or signed) by a truth value. Most commonly, an annotated clause ([35]) has the following form

$$A : \mu \leftarrow B_1 : \mu_1, \ldots, B_n : \mu_n,$$

where each μ_i is an *annotation term*, which means that it is either an annotation constant, or an annotation variable, or a function over annotation constants and/or variables. For ease of implementation, this approach has been very popular and many variations of annotated and signed logic programs have appeared, see [36, 22, 35, 15, 23, 37, 38, 39, 40], for example. We consider logic programs of this type in Section 11.4 of the chapter.

The work of this chapter falls into two main sections: Section 11.3, concerned with annotation-free logic programs, and Section 11.4, concerned with annotated logic programs. However, there is considerable interaction between these two sections, as we show, and the final result, Theorem 4, strongly connects them. However, our approach in Section 11.3 is more algebraic than usual and we develop an abstract, general semantic operator $\mathfrak{T}_{P,\mathfrak{C}}$ defined over certain algebraic structures $\mathfrak{C}$. This theory not only gives a unified view of conventional many-valued logic programming, and of simple logic-based models of uncertainty in logic programming and databases, see [32], but it also includes semiring-based constraint logic programming as considered in [25]; one simply chooses $\mathfrak{C}$ suitably. In defining this operator, we are inspired by Fitting's paper [41] and we work over the set P^{**}, see Section 11.3.1. We are therefore faced with the problem of handling countable (possibly infinite) products $\bigodot_{i\in\mathbb{N}} c_i$, where $\odot$ is a binary operation defined on $\mathfrak{C}$, and with determining the value of $\bigodot_{i\in\mathbb{N}} c_i$ finitely, in some sense. Related problems also arise in constructing the neural networks we present to compute $\mathfrak{T}_{P,\mathfrak{C}}$ when P is propositional. We solve these problems by introducing the notion of finitely-determined binary operations $\odot$, see Definition 1 and Theorem 1, and it turns out that this notion is well-suited to these purposes and to building ANN, somewhat in the style of [1], to compute $\mathfrak{T}_{P,\mathfrak{C}}$, see Theorem 2. In fact, to construct the networks in this case, we introduce the notion of addition and multiplication units which are 2-layer ANN composed of binary threshold units, and this approach produces conceptually quite simple networks.

In Section 11.4, we focus on bilattice-based annotated logic programs and introduce a semantic operator $\mathcal{T}_P$ for them. Again, we build ANN in the style of [1], but this time to simulate $\mathcal{T}_P$. The operator $\mathcal{T}_P$ is simpler than $\mathfrak{T}_{P,\mathfrak{C}}$

to define in so much as it employs P directly, rather than P^{**}, but its simulation by ANN is more difficult than is the case for $\mathfrak{T}_{P,\mathfrak{C}}$. Indeed, the ANN used in Section 11.4 do not require additional layers, but instead employ two learning functions in order to perform computations of $\mathcal{T}_P$. In this sense, one has a tradeoff: more complicated units and simpler connections versus simpler units and more complicated connections. This reduction of architecture and the use of learning functions builds a bridge between the essentially deductive neural networks of [1] and the learning neural networks implemented in Neurocomputing [42]. It also raises questions on the comparative time and space complexity of the architectures of the ANN used to compute $\mathfrak{T}_{P,\mathfrak{C}}$ and $\mathcal{T}_P$, see Section 11.6 summarizing our conclusions, but we do not pursue this issue here in detail.

Finally, we show in Theorem 4 how the computations of $\mathfrak{T}_{P,\mathfrak{C}}$ by the ANN of Section 11.3 can be simulated by computations of $\mathcal{T}_P$ and the corresponding neural networks of Section 11.4. This result unites the two main sections of the chapter and brings it to a conclusion.

For reasons of lack of space, we do not consider here the extension of our results to the case of first-order logic programs P. However, the papers by the present authors listed at the end of the chapter contain some results in this direction.

Acknowlegement The authors thank the Boole Centre for Research in Informatics at University College Cork for its substantial financial support of the research presented here.

11.2 Neural Networks

We begin by briefly summarizing what we need relating to artificial neural networks or just neural networks for short; our terminology and notation are standard, and our treatment closely follows that of [10], but see also [9, 43].

A *neural network* or *connectionist network* is a weighted digraph. A typical *unit* (or node) k in this digraph is shown in Figure 11.1. We let $w_{kj} \in \mathbb{R}$ denote the weight of the connection from unit j to unit k (w_{kj} may be 0). Then the unit k is characterized, at time t, by the following data: its *inputs* $i_{kj}(t) = w_{kj}v_j(t)$ (the input received by k from j at time t) for $j = 1, \ldots, n_k$, its *threshold* $\theta_k \in \mathbb{R}$, its *potential* $p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj}v_j(t)\right) - \theta_k \in \mathbb{R}$, and its *value* $v_k(t)$. The units are updated synchronously, time becomes $t + \Delta t$, and the output value for k, $v_k(t+\Delta t)$, is calculated from $p_k(t)$ by means of a given *output function* ψ, that is, $v_k(t+\Delta t) = \psi(p_k(t))$. The only output function ψ we use here is the Heaviside function H. Thus, $v_k(t + \Delta t) = H(p_k(t))$, where H is defined by $H(x)$ is equal to 1 if $x \geq 0$ and is equal to 0 otherwise. Units of this type are called *binary threshold units*.

As far as the architecture of neural networks is concerned, we will only consider networks where the units can be organized in layers. A *layer* is a

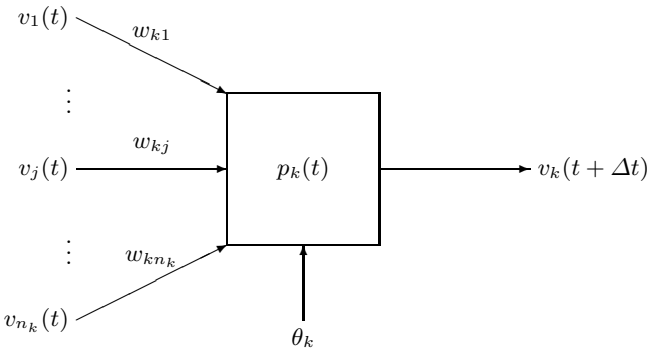


Fig. 11.1. Unit k in a connectionist network.

vector of units. An *n-layer feedforward network* $\mathcal{F}$ consists of the *input* layer, $n-2$ *hidden* layers, and the *output* layer, where $n \geq 2$. Here, we will mainly be concerned with $n = 3$. Each unit occurring in the i-th layer is connected to each unit occurring in the $(i+1)$-st layer, $1 \leq i < n$. A connectionist network $\mathcal{F}$ is called a *multilayer feedforward network* if it is an n-layer feedforward network for some n. Let r and s be the number of units occurring in the input and output layers, respectively, of a multilayer feedforward network $\mathcal{F}$. Then $\mathcal{F}$ computes a function $f_{\mathcal{F}} : \mathbb{R}^r \to \mathbb{R}^s$ as follows. The input vector (the argument of $f_{\mathcal{F}}$) is presented to the input layer at time t_0 and propagated through the hidden layers to the output layer. At each time point, all units update their potential and value. At time $t_0+(n-1)\Delta t$, the output vector (the image under $f_{\mathcal{F}}$ of the input layer) is read off the output layer. Finally, a neural network is called *recurrent* or is *made recurrent* if the number of units in the input layer is equal to the number of units in the output layer and each unit in the output layer is connected with weight 1 to the corresponding unit in the input layer. A recurrent network can thus perform iterated computations because the output values can be returned to the input layer via the connections just described; it can thus perform computation of the iterates $T^k(I)$, $k \in \mathbb{N}$, for example, where I is an interpretation and T is a (semantic) operator.

11.3 Annotation-Free Logic Programs and the Operator $\mathfrak{T}_{P,\mathfrak{C}}$

11.3.1 Finitely-Determined Operations

Let $\mathfrak{C}$ denote a set endowed with binary operations $+$ and $\times$, and a unary operation $\neg$ satisfying $\neg(\neg c) = c$ for all $c \in \mathfrak{C}$[3]. In [44], the notion of *finitely-determined disjunctions and conjunctions* ($\vee$ and $\wedge$) was introduced for sets $\mathfrak{C}$ of truth values. We begin by giving this definition for a general binary operation $\odot$ on $\mathfrak{C}$. Note that we assume that $\odot$ has meaningfully been extended to include products $\bigodot_{i\in M} c_i$ of countably infinite families M of elements c_i of $\mathfrak{C}$. Indeed, the way in which we carry out this extension is the main point of the next definition and the discussion following it.

Definition 1. *Suppose that $\mathfrak{C}$ is a set equipped with a binary operation $\odot$. We say that $\odot$ is* finitely determined *or that* products (relative to $\odot$) are finitely determined in $\mathfrak{C}$ *if, for each $c \in \mathfrak{C}$, there exists a countable (possibly infinite) collection $\{(R^n_c, E^n_c) \mid n \in \mathcal{J}\}$ of pairs of sets $R^n_c \subseteq \mathfrak{C}$ and $E^n_c \subseteq \mathfrak{C}$, where each R^n_c is finite, such that a countable (possibly infinite) product $\bigodot_{i\in M} c_i$ in $\mathfrak{C}$ is equal to c if and only if for some $n \in \mathcal{J}$ we have*
(i) $R^n_c \subseteq \{c_i \mid i \in M\}$, and
(ii) for all $i \in M$, $c_i \notin E^n_c$, that is, $\{c_i \mid i \in M\} \subseteq (E^n_c)^{co}$, where $(E^n_c)^{co}$ denotes the complement of the set E^n_c.

We call the elements of E^n_c excluded values *for c, we call the elements of $\mathcal{A}^n_c = (E^n_c)^{co}$* allowable values *for c, and in particular we call the elements of R^n_c* required values *for c; note that, for each $n \in \mathcal{J}$, we have $R^n_c \subseteq \mathcal{A}^n_c$, so that each required value is also an allowable value (but not conversely). More generally, given $c \in \mathfrak{C}$, we call $s \in \mathfrak{C}$ an* excluded value for c *if no product $\bigodot_{i\in M} c_i$ with $\bigodot_{i\in M} c_i = c$ contains s, that is, in any product $\bigodot_{i\in M} c_i$ whose value is equal to c, we have $c_i = s$ for no $i \in M$. We let E_c denote the set of all excluded values for c, and let $\mathcal{A}_c$ denote the complement $(E_c)^{co}$ of E_c and call it the set of all* allowable values *for c. Note finally that when confusion might otherwise result, we will superscript each of the sets introduced above with the operation in question. Thus, for example, $\mathcal{A}^{\odot}_c$ denotes the allowable set for c relative to the operation $\odot$.*

This definition was originally motivated by the results of [45], and the following example shows the thinking behind it.

Example 1. Consider Belnap's well-known four-valued logic with set $\mathfrak{C} = \mathcal{FOUR} = \{t, u, b, f\}$ of truth values and connectives as defined in Table 11.1, where t denotes *true*, u denotes *undefined* or *none* (neither true nor false), b denotes *both* (true and false), and f denotes *false*.

Taking $\odot$ to be disjunction $\vee$, the sets E and R are as follows.
(a) For t, n takes values 1 and 2, $E^{\vee}_t = \emptyset$, $R^{\vee,1}_t = \{t\}$, and $R^{\vee,2}_t = \{u, b\}$.

[3] When $\mathfrak{C}$ has no natural negation $\neg$, we simply take $\neg$ to be the identity.

Table 11.1. Truth table for the logic $\mathcal{FOUR}$

p	q	$\neg p$	$p \wedge q$	$p \vee q$
t	t	f	t	t
t	u	f	u	t
t	b	f	b	t
t	f	f	f	t
u	t	u	u	t
u	u	u	u	u
u	b	u	f	t
u	f	u	f	u
b	t	b	b	t
b	u	b	f	t
b	b	b	b	b
b	f	b	f	b
f	t	t	f	t
f	u	t	f	u
f	b	t	f	b
f	f	t	f	f

(b) For u, we have $n = 1$, $E_u^\vee = \{t, b\}$ and $R_u^\vee = \{u\}$.
(c) For b, we have $n = 1$, $E_b^\vee = \{t, u\}$ and $R_b^\vee = \{b\}$.
(d) For f, we have $n = 1$, $E_f^\vee = \{t, u, b\}$ and $R_f^\vee = \{f\}$.

Thus, a countable disjunction $\bigvee_{i \in M} c_i$ takes value t if and only if either (i) at least one of the c_i is equal to t or (ii) at least one of the c_i takes value b and at least one takes value u; no truth value is excluded. As another example, $\bigvee_{i \in M} c_i$ takes value u if and only if at least one of the c_i is u, none are equal to t and none are equal to b.

Now taking $\odot$ to be conjunction $\wedge$, the sets E and R are as follows.
(a) For t, we have $n = 1$, $E_t^\wedge = \{u, b, f\}$ and $R_t^\wedge = \{t\}$.
(b) For u, we have $n = 1$, $E_u^\wedge = \{b, f\}$ and $R_u^\wedge = \{u\}$.
(c) For b, we have $n = 1$, $E_b^\wedge = \{u, f\}$ and $R_b^\wedge = \{b\}$.
(d) For f, n takes values 1 and 2, $E_f^\wedge = \emptyset$, $R_f^{\wedge,1} = \{f\}$, and $R_f^{\wedge,2} = \{u, b\}$.

Notice finally that, if we restrict the connectives in $\mathcal{FOUR}$ to $\{t, u, f\}$, we obtain Kleene's well-known strong three-valued logic.

It turns out that the connectives in all the logics commonly encountered in logic programming, and indeed in many other logics, satisfy Definition 1, and it will be convenient to state next the main facts we need concerning arbitrary finitely-determined operations, see [44] for all proofs.

Theorem 1. *Suppose that $\odot$ is a binary operation defined on a set $\mathfrak{C}$. Then the following statements hold.*

1. *If $\odot$ is finitely determined, then it is idempotent, commutative and associative.*

2. *Suppose that $\odot$ is finitely determined and that $\mathfrak{C}$ contains finitely many elements $\{c_1, \ldots, c_n\}$. Then, for any collection $\{s_i \mid i \in M\}$, where each of the $s_i \in \mathfrak{C}$ and M is a denumerable set, the sequence $s_1, s_1 \odot s_2, s_1 \odot s_2 \odot s_3, \ldots$ is eventually constant with value s, say. Therefore, setting $\bigodot_{i \in M} s_i = s$ gives each countably infinite product in $\mathfrak{C}$ a well-defined meaning which extends the usual meaning of finite products.*
3. *Suppose $\odot$ is finitely determined and that $\bigodot_{i \in M} s_i = c$, where M is a countable set. Then the sequence $s_1, s_1 \odot s_2, s_1 \odot s_2 \odot s_3, \ldots$ is eventually constant with value c.*
4. *Suppose that $\mathfrak{C}$ is a countable set and $\odot$ is idempotent, commutative and associative. Suppose further that, for any set $\{s_i \mid i \in M\}$ of elements of $\mathfrak{C}$ where M is countable, the sequence $s_1, s_1 \odot s_2, s_1 \odot s_2 \odot s_3, \ldots$ is eventually constant. Then all products in $\mathfrak{C}$ are (well-defined and are) finitely determined.*
5. *Suppose that $\mathfrak{C}$ is finite. Then $\odot$ is finitely determined if and only if it is idempotent, associative and commutative.*

For a finitely-determined binary operation $\odot$ on $\mathfrak{C}$, we define the partial order $\leq_\odot$ on $\mathfrak{C}$ by $s \leq_\odot t$ if and only if $s \odot t = t$. (So that $s \leq_+ t$ if and only if $s + t = t$, and $s \leq_\times t$ if and only if $s \times t = t$, for finitely-determined operations $+$ and $\times$.) Note (i) that the orderings $\leq_+$ and $\leq_\times$ are dual to each other if and only if the absorption law holds for $+$ and $\times$, in which case $(\mathfrak{C}, \leq_+, \leq_\times)$ is a lattice, and (ii) that finitely-determined operations $+$ and $\times$ need not satisfy the distributive laws. Notice also that because $+$ and $\times$ are finitely determined, $\sum_{c \in \mathfrak{C}} c \in \mathfrak{C}$ is the top element of $\mathfrak{C}$ relative to $\leq_+$, and $\prod_{c \in \mathfrak{C}} c \in \mathfrak{C}$ is the top element of $\mathfrak{C}$ relative to $\leq_\times$. Note, however, that it does not follow that we have bottom elements for these orderings. We further suppose that two elements $\overline{c}$ and $\underline{c}$ are distinguished in $\mathfrak{C}$, and we will make use of these elements later on. (In some, but not all, situations when $\mathfrak{C}$ is a logic, $\overline{c}$ is taken to be *true*, and $\underline{c}$ is taken to be *false*.)

Example 2. In $\mathcal{FOUR}$, we have $t \leq_\wedge u \leq_\wedge f$, and $t \leq_\wedge b \leq_\wedge f$. Also, $f \leq_\vee u \leq_\vee t$, and $f \leq_\vee b \leq_\vee t$. In this case, we take $\overline{c} = t$, and $\underline{c} = f$.

Although we will not need to suppose here that $\mathfrak{C}$ is a complete partial order in the orders just defined, that assumption is often made and then a least element with respect to $\leq_+$ must be present in $\mathfrak{C}$ (we add this element to $\mathfrak{C}$ if necessary). In particular, to calculate the least fixed point of $\mathfrak{T}_{P,\mathfrak{C}}$, it is common practice to iterate on the least element. However, if we require the least fixed point to coincide with any useful semantics, it will usually be necessary to choose the default value $\underline{c} \in \mathfrak{C}$ to be the least element in the ordering $\leq_+$.

Furthermore, the allowable and excluded sets for $s \in \mathfrak{C}$ can easily be characterized in terms of these partial orders: $s \in \mathcal{A}_t^\odot$ if and only if $s \leq_\odot t$, see [44, Proposition 3.10]. Because of this fact, the following result plays an important role in the construction of the network to compute $\mathfrak{T}_{P,\mathfrak{C}}$, as we see later.

Proposition 1. *Suppose that $\odot$ is a finitely-determined binary operation on $\mathfrak{C}$ and that M is a countable set. Then a product $\bigodot_{i\in M} t_i$ evaluates to the element $s \in \mathfrak{C}$, where s is the least element in the ordering $\leq_\odot$ such that $t_i \in \mathcal{A}_s^\odot$ for all $i \in M$.*

Proof. Assume that $\bigodot_{i\in M} t_i = s$. Then each t_i is an allowable value for s and so we certainly have $t_i \in \mathcal{A}_s^\odot$ for all $i \in M$.

Now assume that $\{t_i \mid i \in M\} \subseteq \mathcal{A}_t^\odot$; we want to show that $s = \bigodot_{i\in M} t_i \in \mathcal{A}_t^\odot$, for then it will follow that $s \leq_\odot t$, as required. By Theorem 1, we may suppose that M is finite, $M = \{1, \ldots, n\}$, say. Since $t_i \leq_\odot t$ for $i = 1, \ldots, n$, we have $t_i \odot t = t$ for $i = 1, \ldots, n$. Therefore, by Statement 1 of Theorem 1, we obtain $t = \bigodot_{i\in M}(t_i \odot t) = (\bigodot_{i\in M} t_i) \odot t = s \odot t$. It follows that $s \leq_\odot t$, and the proof is complete.

In what follows throughout this section, $\mathfrak{C}$ will denote a set endowed with binary operations $+$ and $\times$; furthermore, $+$ at least will be assumed to be finitely determined and $\times$ will be assumed to be associative for simplicity. In § 11.3.3 and § 11.3.4, we will also need to assume that $\times$ is finitely determined. In fact, it transpires that our main definition (but not all our results) can be made simply in the context of the set $\mathfrak{C}$ with sufficient completeness properties, namely, that arbitrary countable sums can be defined.

Of particular interest to us are the following three cases.
(1) $\mathfrak{C}$ is a set of truth values, $+$ is disjunction $\vee$ and $\times$ is conjunction $\wedge$.
(2) $\mathfrak{C}$ is a *c*-semiring (constraint-based semiring) as considered in [25]. Thus, $\mathfrak{C}$ is a semiring, where the top element in the order $\leq_\times$ is the identity element $\mathbf{0}$ for $+$, and the top element in the order $\leq_+$ is the identity element $\mathbf{1}$ for $\times$. In addition, $+$ is idempotent, $\times$ is commutative, and $\mathbf{1}$ annihilates $\mathfrak{C}$ relative to $+$, that is, $\mathbf{1} + c = c + \mathbf{1} = \mathbf{1}$ for all elements $c \in \mathfrak{C}$.
(3) $\mathfrak{C}$ is the set L_m of truth values considered in [32], $+$ is max and $\times$ is min. These will be discussed briefly in Example 3.

11.3.2 The operator $\mathfrak{T}_{P,\mathfrak{C}}$

We next turn to giving the definition of the operator $\mathfrak{T}_{P,\mathfrak{C}}$.

Let $\mathcal{L}$ be a first-order language, see [46] for notation and undefined terms relating to conventional logic programming, and suppose that $\mathfrak{C}$ is given. By a $\mathfrak{C}$-*normal logic program* P or a *normal logic program* P *defined over* $\mathfrak{C}$, we mean a finite set of clauses or rules of the type $A \leftarrow L_1, \ldots, L_n$ (n may be 0, by the usual abuse of notation), where A is an atom in $\mathcal{L}$ and the L_j, for $1 \leq j \leq n$, are either literals in $\mathcal{L}$ or are elements of $\mathfrak{C}$. By a $\mathfrak{C}$-*interpretation* or just *interpretation* I for P, we mean a mapping $I : B_P \to \mathfrak{C}$, where B_P denotes the Herbrand base for P. We immediately extend I to $\neg \cdot B_P$ by $I(\neg A) = \neg I(A)$, for all $A \in B_P$, and to $B_P \cup \neg \cdot B_P \cup \mathfrak{C}$ by setting $I(c) = c$ for all $c \in \mathfrak{C}$. Finally, we let $I_{P,\mathfrak{C}}$ or simply I_P denote the set of all $\mathfrak{C}$-interpretations for P ordered by $\sqsubseteq_+$, that is, by the pointwise ordering relative to $\leq_+$. Notice

that the value $I(L_1, \ldots, L_n)$ of I on any clause body is uniquely determined[4] by $I(L_1, \ldots, L_n) = I(L_1) \times \ldots \times I(L_n)$.

To define the *semantic operator* $\mathfrak{T}_{P,\mathfrak{C}}$, we essentially follow [41], allowing for our extra generality, in first defining the sets P^* and P^{**} associated with P. To define P^*, we first put in P^* all ground instances of clauses of P whose bodies are non-empty. Second, if a clause $A \leftarrow$ with empty body occurs in P, add $A \leftarrow \overline{c}$ to P^*. Finally, if the ground atom A is not yet the head of any member of P^*, add $A \leftarrow \underline{c}$ to P^*. To define P^{**}, we note that there may be many, even denumerably many, elements $A \leftarrow C_1$, $A \leftarrow C_2$, ... of P^* having the same head A. We replace them with $A \leftarrow C_1 + C_2 + \ldots$, where $C_1 + C_2 + \ldots$ is to be thought of as a formal sum. Doing this for each A gives us the set P^{**}. Now, each ground atom A is the *head* of exactly one element $A \leftarrow C_1 + C_2 + \ldots$ of P^{**}, and it is common practice to work with P^{**} in place of P. Indeed, $A \leftarrow C_1 + C_2 + \ldots$ may be written $A \leftarrow \sum_i C_i$ and referred to as a (or as the) *pseudo-clause* with *head* A and *body* $\sum_i C_i$.

Definition 2. *(See [41]) Let P be a $\mathfrak{C}$-normal logic program. We define $\mathfrak{T}_{P,\mathfrak{C}} : I_{P,\mathfrak{C}} \to I_{P,\mathfrak{C}}$ as follows. For any $I \in I_{P,\mathfrak{C}}$ and $A \in B_P$, we set*

$$\mathfrak{T}_{P,\mathfrak{C}}(I)(A) = I(\textstyle\sum_i C_i) = \sum_i I(C_i),$$

where $A \leftarrow \sum_i C_i$ is the unique pseudo-clause in P^{**} whose head is A. Note that when $\mathfrak{C}$ is understood, we may denote $\mathfrak{T}_{P,\mathfrak{C}}$ simply by $\mathfrak{T}_P$.

We note that $I(\sum_i C_i) = \sum_i I(C_i)$ is well-defined in $\mathfrak{C}$ by Theorem 1. Indeed, $\sum_i I(C_i)$ may be a denumerable sum in $\mathfrak{C}$, and it is this observation which motivates the introduction of the notion of finite determinedness.

Example 3. Some special cases of $\mathfrak{T}_{P,\mathfrak{C}}$. As mentioned in the introduction, the operator $\mathfrak{T}_{P,\mathfrak{C}}$ includes a number of important cases simply by choosing $\mathfrak{C}$ suitably, and we briefly consider this point next.
(1) The standard semantics of logic programming. Choosing $\mathfrak{C}$ to be classical two-valued logic, Kleene's strong three-valued logic, and $\mathcal{FOUR}$, one recovers respectively the usual single-step operator T_P, Fitting's three-valued operator Φ_P, and Fitting's four-valued operator Ψ_P, see [41]. Hence, one recovers the associated semantics as the least fixed points of $\mathfrak{T}_{P,\mathfrak{C}}$.

Furthermore, in [47], Wendt studied the fixpoint completion, fix(P), of a normal logic program P introduced by Dung and Kanchanasut in [48]. The fixpoint completion is a normal logic program in which all body literals are negated, and is obtained by a complete unfolding of the recursion through positive literals in the clauses of a program. In fact, Wendt obtained interesting connections between various semantic operators by means of fix(P). Specifically, he showed that for any normal logic program P, we have (i) $GL_P(I) = T_{\mathrm{fix}(P)}(I)$ for any two-valued interpretation I, and (ii)

[4] This is meaningful even if $\times$ is not associative provided bracketing is introduced and respected.

$\overline{\Psi}_P(I) = \Phi_{\mathrm{fix}(P)}(I)$ for any three-valued interpretation I, where GL_P is the well-known operator of Gelfond and Lifschitz used in defining the stable-model semantics, and $\overline{\Psi}_P$ is the operator used in [49] to characterize the well-founded semantics of P. These connections have the immediate corollary that GL_P and $\overline{\Psi}_P$ can be seen as special cases of $\mathfrak{T}_{P,\mathfrak{C}}$, and hence that the well-founded and stable-model semantics can be viewed as special cases of the fixed points of $\mathfrak{T}_{P,\mathfrak{C}}$.
(2) Constraint logic programs. A *semiring-based constraint logic program* P, see [25], consists of a finite set of clauses each of which is of the form

$$A \leftarrow L_1, L_2, \ldots, L_k, \tag{11.1}$$

where A is an atom and the L_i are literals or is of the form

$$A \leftarrow a, \tag{11.2}$$

where A is an atom and a is any semiring value. Those clauses with a semiring value in the body constitute the constraints and are also known as "facts". The distinguished values $\overline{c}$ and $\underline{c}$ in a c-semiring are $\mathbf{1}$ and $\mathbf{0}$ respectively. Thus, when constructing P^* for a semiring-based constraint logic program P, unit clauses $A \leftarrow$ are replaced by $A \leftarrow \mathbf{1}$ and for any atom A not the head of a clause, we add the clause $A \leftarrow \mathbf{0}$ to P^*.

In this context, an interpretation I is a mapping $I : B_P \rightarrow \mathcal{S}$, where $\mathcal{S} = \mathfrak{C}$ is the underlying c-semiring, and we denote by $I_{P,\mathcal{S}}$ the set of all such interpretations. Finally, associated with each semiring-based constraint logic program P is a consequence operator $T_{P,\mathcal{S}} : I_{P,\mathcal{S}} \rightarrow I_{P,\mathcal{S}}$ defined in [25] essentially as follows.

Definition 3. *Given an interpretation I and a ground atom A, we define $T_{P,\mathcal{S}}(I)$ by*

$$T_{P,\mathcal{S}}(I)(A) = \textstyle\sum_i I(C_i),$$

where $A \leftarrow \sum_i C_i$ is the unique pseudo-clause whose head is A, and $I(C_i)$ is defined as follows. We set $I(C_i) = a$ when $A \leftarrow \sum_i C_i$ is the fact $A \leftarrow a$, and otherwise when $A \leftarrow \sum_i C_i$ is not a fact of the form $A \leftarrow a$, we set $I(C_i) = \prod_{j=1}^{n_i} I(L_j^i)$, where $C_i = L_1^i, \ldots, L_{n_i}^i$, say.

It is easy to see that if P is a semiring-based constraint logic program, then the semantic operator $\mathfrak{T}_{P,\mathfrak{C}}$ coincides with $T_{P,\mathcal{S}}$ when we take $\mathfrak{C}$ to be the c-semiring $\mathcal{S}$ underlying P, as already observed.

Another example of interest in this context concerns uncertainty in rule-based systems, such as those considered in [32], but we omit the details.

11.3.3 Towards the Computation of $\mathfrak{T}_{P,\mathfrak{C}}$ by ANN

As noted in the introduction, an algorithm is presented in [1] for constructing an ANN which computes T_P exactly for any given propositional logic program

P. Indeed, it is further shown in [1] that 2-layer binary threshold feedforward networks cannot do this. In the algorithms discussed in [1], single units hold the truth values of atoms and clauses in P. A unit outputs 1 if the corresponding atom/clause is true with respect to the interpretation presented to the network, and outputs 0 if the corresponding atom/clause is false. However, when dealing with many-valued logics or with sets $\mathfrak{C}$ with more than two elements, single units can compute neither products nor sums of elements.

In this subsection and the next, we discuss the computation by ANN of the semantic operator $\mathfrak{T}_{P,\mathfrak{C}}$ determined by a propositional normal logic program P defined over some set $\mathfrak{C}$. This is done by simulating the clauses and the connections between the body literals in the clauses of P. This requires that we represent elements of $\mathfrak{C}$ by units, or combinations of them, in ANN and then simulate the operations of $+$ and $\times$ in $\mathfrak{C}$ by computations in ANN. To keep matters simple, and since we are not employing learning algorithms here, we shall focus our attention on binary threshold units only.

In order to define $\mathfrak{T}_{P,\mathfrak{C}}$, it is not even necessary for multiplication to be commutative because only finite products occur in the bodies of elements of P and P^{**}. Nevertheless, it will be necessary here in computing $\mathfrak{T}_{P,\mathfrak{C}}$ by ANN to impose the condition that multiplication is in fact finitely determined, since we need to make use of Proposition 1 relative to both addition and multiplication.

We shall focus on finite sets $\mathfrak{C}$ with n elements listed in some fixed order, $\mathfrak{C} = \{c_1, c_2, \ldots, c_n\}$ or $\mathfrak{C} = \{t_1, t_2, \ldots, t_n\}$, say. In order to simulate the operations in $\mathfrak{C}$ by means of ANN, we need to represent the elements of $\mathfrak{C}$ in a form amenable to their manipulation by ANN. To do this, we represent elements of $\mathfrak{C}$ by vectors of n units[5], where the first unit represents c_1, the second unit represents c_2, and so on. Hence, a vector of units with the first unit activated, or containing 1, represents c_1, a vector with the second unit activated, or containing 1, represents c_2, etc. Indeed, it will sometimes be convenient to denote such vectors by binary strings of length n, and to refer to the unit in the i-th position of a string as the i-th unit or the c_i-unit or the unit c_i; as is common, we represent these vectors geometrically by strings of not-necessarily adjacent rectangles. Note that we do not allow more than one unit to be activated at any given time in any of the vectors representing elements of $\mathfrak{C}$, and hence all but one of the units in such vectors contain 0. Furthermore, when the input is consistent with this, we shall see from the constructions we make that the output of any network we employ is consistent with it also.

Example 4. Suppose that $\mathfrak{C} = \mathcal{FOUR} = \{t, u, b, f\}$, listed as shown. Then t is represented by 1000, u by 0100, b by 0010, and f by 0001.

In general, the operations in $\mathfrak{C}$ are not linearly separable, and therefore we need two layers to compute addition and two to compute multiplication. As usual, we take the standard threshold for binary threshold units to be

[5] It is convenient sometimes to view them as column vectors.

0.5. This ensures that the Heaviside function outputs 1 if the input is strictly greater than 0, rather than greater than or equal to 0.

Definition 4. *A* multiplication ($\times$) unit *or a* conjunction ($\wedge$) unit $\mathcal{MU}$ *for a given set* $\mathfrak{C}$ *is a* 2*-layer ANN in which each layer is a vector of* n *binary threshold units* $c_1, c_2, \ldots, c_n$ *corresponding to the* n *elements of* $\mathfrak{C}$*. The units in the input layer have thresholds* $l - 0.5$*, where* l *is the number of elements being multiplied or conjoined, and all output units have threshold* 0.5*. We connect input unit* c_i *to the output unit* c_i *with weight* 1 *and to any unit* c_j *in the output layer, where* $c_i <_\times c_j$*, with weight* -1*.*

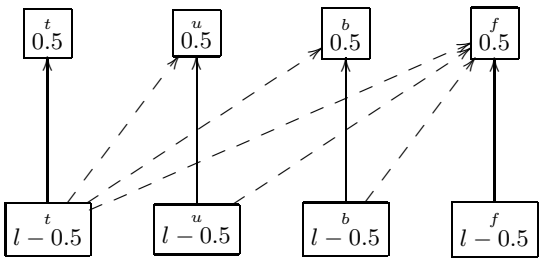


Fig. 11.2. A conjunction unit for $\mathcal{FOUR}$. The full arrows represent connections with weight 1, and the broken arrows represent connections with weight -1.

An input layer representing a product of l elements of $\mathfrak{C}$ is connected to a multiplication unit $\mathcal{MU}$ in the following way. For each element c of the product, where c is represented by the n units $c_1, c_2, \ldots, c_n$, the unit c_j is connected, with weight 1, to the c_j-unit in the input layer of $\mathcal{MU}$ and is also connected, with weight 1, to any unit c_k in the input layer of $\mathcal{MU}$ for which $c_j <_\times c_k$. For a negated element $d = \neg c$ in the product, we connect, with weight 1, c_j to the unit representing $\neg c_j$ in the input layer of $\mathcal{MU}$ and also, with weight 1, to any unit c_k in the input layer of $\mathcal{MU}$ for which $\neg c_j <_\times c_k$.

Proposition 2. *A multiplication or conjunction unit* $\mathcal{MU}$ *computes the value of a product or conjunction of* l *elements of* $\mathfrak{C}$ *when it is connected to an input layer as just described.*

Proof. The proof ultimately depends on Proposition 1 and indeed $\mathcal{MU}$, in effect, counts the number of elements of $\mathfrak{C}$ which are in the allowable set $\mathcal{A}_s^\times$ for each $s \in \mathfrak{C}$. Suppose given a product $s_1 \times s_2 \times \cdots \times s_l$, of l elements of $\mathfrak{C}$, whose value is equal to $c_i \in \mathfrak{C}$, so that each of the elements in this product is an allowable value for c_i. By the manner of connecting the units representing the s_k to the input layer of $\mathcal{MU}$, the c_j-unit in the input layer of $\mathcal{MU}$ will be activated if and only if $c_i \leq_\times c_j$. By construction of $\mathcal{MU}$, any c_j-unit in the output layer of $\mathcal{MU}$ for which the c_j-unit in the input layer is activated will

receive an input of 1 ($= 1 \times 1$). However, any c_j-unit in the output layer of $\mathcal{MU}$ for which $c_i <_\times c_j$ will also receive negative input whereas the unit c_i itself will receive no such input. Therefore, the unit c_i is the only unit activated in the output layer of $\mathcal{MU}$, as required.

Example 5. Consider again $\mathfrak{C} = \mathcal{FOUR} = \{t, u, b, f\}$, and input the two elements u and b to a multiplication unit $\mathcal{MU}$, where $l = 2$. It is readily checked that the potentials of the units t, u and b in the input layer of $\mathcal{MU}$ are respectively -1.5, -0.5 and -0.5, that their outputs are all equal to 0, and that the outputs of the units t, u and b in the output layer of $\mathcal{MU}$ are also all equal to 0. On the other hand, the f-unit in the input layer of $\mathcal{MU}$ has potential $1 \times 0 + 1 \times 0 + 1 \times 1 + 1 \times 0 + 1 \times 0 + 1 \times 1 + 1 \times 0 + 1 \times 0 - 1.5 = 0.5$, and therefore the output of this unit is $H(0.5) = 1$. Furthermore, the input to the f-unit in the output layer of $\mathcal{MU}$ is $-1 \times 0 - 1 \times 0 - 1 \times 0 + 1 \times 1 = 1$. Hence, the output of this unit is $H(1 - 0.5) = 1$, and so $\mathcal{MU}$ outputs 0001 or f and this indeed is the value of $u \wedge b$, as required.

Note 1. (1) If we take $l = 1$, that is, if we consider a product of just one element c of $\mathfrak{C}$, then a multiplication unit outputs c whenever c is input to that unit. The same comment applies to addition units also, and these observations will be used in the network we construct in Theorem 2 to handle clauses whose body contains just one element.
(2) Suppose that $\mathcal{MU}$ is a multiplication unit with $l \geq 2$ and that $c \in \mathfrak{C}$. By permanently connected a vector representing c to $\mathcal{MU}$, we obtain a multiplication unit $\mathcal{MU}(c)$ which multiplies any input to it by c. We shall refer to $\mathcal{MU}$ as a *multiplication unit with one factor fixed at c*. One can similarly construct multiplication units with any number of factors fixed at elements of $\mathfrak{C}$. Such units will be needed in Step 2.2 of the translation algorithm used in Theorem 2.

The ideas behind multiplication units work, with minor changes, for addition or disjunction, and we consider this point next.

Definition 5. *An* addition (+) unit *or a* disjunction ($\vee$) unit $\mathcal{AU}$ *for a given set* $\mathfrak{C}$ *is a* 2*-layer ANN in which each layer is a vector of* n *binary threshold units* $c_1, c_2, \ldots, c_n$ *corresponding to the* n *elements of* $\mathfrak{C}$*. The units in the input layer have threshold* $k - 0.5$*, where* k *is the number of elements to be added or disjoined, and all output units have threshold* 0.5*. We connect input unit* c_i *to the output unit* c_i *with weight* 1 *and to any unit* c_j *in the output layer, where* $c_i <_+ c_j$*, with weight* -1*.*

The manner of connecting an input layer to an addition unit, and the calculation of a sum or disjunction of elements proceeds exactly as for multiplication, and again in effect makes use of Proposition 1.

Proposition 3. *An addition or disjunction unit* $\mathcal{AU}$ *computes the value of a sum or disjunction of* k *elements of* $\mathfrak{C}$ *when it is connected to an input layer as just described.*

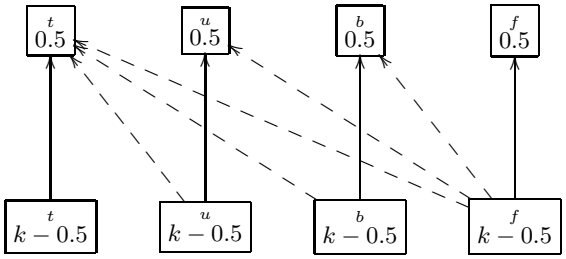


Fig. 11.3. A disjunction unit for $\mathcal{FOUR}$. The full arrows represent connections with weight 1, and the broken arrows represent connections with weight -1.

Proof. The proof follows that of Proposition 2 with the necessary minor changes.

Therefore, both operations in $\mathfrak{C}$ can be simulated by ANN. However, there are occasions when we desire the addition and multiplication units to be independent of the number of elements being added or multiplied: for instance, if we wish to add more clauses after the network has been constructed. To handle this situation, "constant threshold" units for an arbitrary finitely-determined $\mathfrak{C}$ were introduced and studied in [50], but they will not be considered here.

When clauses of the form $A \leftarrow c$, where $c \in \mathfrak{C}$, are present in the programs we consider, we need to compute c in the middle layer of the networks of Theorem 2, and this leads to the introduction of $\mathfrak{C}$-element units.

Definition 6. *A* $\mathfrak{C}$-element unit *for* $c \in \mathfrak{C}$ *is an ANN whose layers and connections are the same as those in a multiplication unit except for the thresholds. The thresholds for all units in the input layer are taken to be equal to* 1 *apart from the unit representing the element* c*, which has threshold taken to be* -0.5*. All thresholds in the output layer are taken to be* 0.5*.*

Note 2. Given any input $a_1, a_2, \ldots, a_n$ to a $\mathfrak{C}$-element unit for c, where all the a_i are either 0 or 1 and only one of them is equal to 1, the unit outputs c. These units will also be needed in Step 2.2 of the translation algorithm of Theorem 2.

Example 6. When $A \leftarrow u$ is present in a program defined over $\mathcal{FOUR}$, the following unit is placed in the middle layer of the networks required to compute $\mathfrak{T}_{P,\mathfrak{C}}$, see Figure 11.4. Only u will be activated at any time and the unit will always output 0100, or u, as desired.

11.3.4 Networks for the Computation of $\mathfrak{T}_{P,\mathfrak{C}}$

Suppose that P is a propositional logic program defined over $\mathfrak{C}$. Then B_P is finite with m elements $\{A_1, A_2, \ldots, A_m\}$, say. However, we need to fix

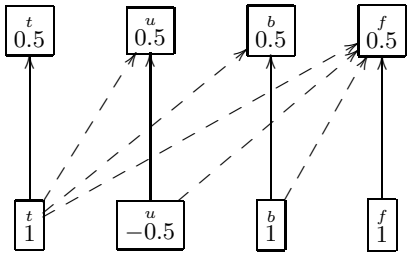


Fig. 11.4. u-unit for $\mathcal{FOUR}$.

an order on the elements of B_P, and henceforth will write B_P as the list $(A_1, A_2, \ldots, A_m)$. For finite sets $\mathfrak{C}$ of cardinality n, we can view interpretations $I \in I_{P,\mathfrak{C}}$ as vectors of length m in which each entry is itself a vector or binary string of length n holding the value $I(A)$ of a particular atom A. Thus, the j-th entry corresponds to A_j in the sense that the string in question will have 1 in its i-th place if $I(A_j) = c_i$, and 0 otherwise. It follows that the input layer of the network we are about to construct contains $m \times n$ units: the first n units correspond to the values of the first atom A_1, the second n correspond to the values of the second atom A_2, and so on.

Example 7. Take $\mathfrak{C}$ to be $\mathcal{FOUR} = \{t, u, b, f\}$, again ordered as listed, and suppose that $B_P = (A_1, A_2, A_3, A_4, A_5)$ has five elements. Then the following vector $(0100, 1000, 0001, 0100, 0010)$ represents a four-valued interpretation I such that $I(A_1) = u$, $I(A_2) = t$, $I(A_3) = f$, $I(A_4) = u$, and $I(A_5) = b$.

We are now in a position to present the following theorem.

Theorem 2. *Suppose that both operations in $\mathfrak{C}$ are finitely determined and that P is a propositional logic program defined over $\mathfrak{C}$. Then we can construct a 3-layer feedforward ANN $\mathcal{F}$ which contains multiplication units in its middle layer and addition units in its output layer such that $\mathcal{F}$ computes $\mathfrak{T}_{P,\mathfrak{C}}$.*

Proof. Because of the preponderance of symbols "c" in the proof, it will ease notation slightly to take $\mathfrak{C}$ as the set $\{t_1, t_2, \ldots, t_n\}$ instead of $\{c_1, c_2, \ldots, c_n\}$.

With the notation already established, let K be the number of clauses in P^*.

1. Set the first and third layers to be vectors of length m. Each entry in the vector in the first layer is itself a vector of n binary threshold units representing an element of B_P, and each entry in the third layer is an addition unit corresponding to an element of B_P. Each unit in each vector corresponds to the (truth) value of an atom $A_j \in B_P$, as described above.
2. For the second layer:
   2.1 Set the second layer to be a vector of length K, with each entry determining the value of the corresponding clause body C by means of a multiplication unit with input layer $C^{t_1}, C^{t_2}, \ldots, C^{t_n}$.

2.2 For each clause in P^*, connect each atom B in its body, C, with weight 1, as follows. Connect the unit B^{t_j} in the first layer of the network to the unit C^{t_j} in the input layer of the multiplication unit corresponding to C, and to any unit C^s in the input layer of that same multiplication unit for which $t_j <_\times s$, where $s \in \mathfrak{C}$. For a negated literal $L = \neg B$ in the body C of the given clause, connect, with weight 1, the unit B^{t_j} in the first layer of the network to the unit $C^{\neg t_j}$ in the input layer of the multiplication unit corresponding to C, and, also with weight 1, to any unit C^s in the input layer of that same multiplication unit for which $\neg t_j <_\times s$, where $s \in \mathfrak{C}$. Note that the connections specified here are precisely those given earlier for connecting a layer to a multiplication unit. Furthermore:
(i) If the body C of the given clause contains elements c of $\mathfrak{C}$ as well as atoms B or literals L, then the corresponding multiplication unit is a multiplication unit with one factor fixed at c for each such c, see Note 1.
(ii) If the given clause is of the form $A \leftarrow c$ for some $c \in \mathfrak{C}$, and hence contains no atoms nor literals, the the corresponding "multiplication unit" is a $\mathfrak{C}$-element unit for c, see Note 2. This unit is connected as a multiplication unit to any vector in the first layer of the network representing an element of B_P.

3. For the clause $A \leftarrow C$ in P^*, connect the unit C^{t_j} in the output layer of the second layer of the network, with weight 1, to the unit A^{t_j} in the input layer of the addition unit corresponding to A in the third layer of the network, and, also with weight 1, to any unit A^s in the input layer of that same addition unit for which $t_j <_+ s$, where $s \in \mathfrak{C}$. Note that the connections specified here are precisely those given earlier for connecting a layer to an addition unit.

Suppose that the interpretation I is presented to the input layer, and that $\mathfrak{T}_{P,\mathfrak{C}}(I)(A) = s$. Then $I(\sum_{i=1}^{h} C_i) = s$, where $A \leftarrow \sum_{i=1}^{h} C_i$ is the unique pseudo-clause in P^{**} with head A. Thus, $I(C_i) \in \mathcal{A}_s^+$ for all $1 \le i \le h$, as in Definition 1. In particular, there exists j with $R_s^{+,j} \subseteq \{I(C_i) \mid 1 \le i \le h\}$. If $R_s^{+,j} = \{r_1, r_2, \ldots, r_g\}$, then there are clauses $A \leftarrow C_{i_1}, A \leftarrow C_{i_2}, \ldots, A \leftarrow C_{i_g}$ in P^* such that $I(C_{i_u}) = r_u$, for $1 \le u \le g$. For each of these $C_{i_u} = L_1, L_2, \ldots, L_l$, say, we have $I(L_i) \in \mathcal{A}_{r_u}^\times$. Therefore, there is a j' such that $R_{r_u}^{\times,j'} = \{q_1, q_2, \ldots, q_v\} \subseteq \mathcal{A}_{r_u}^\times$ and $I(L_{\lambda_1}) = q_1$, $I(L_{\lambda_2}) = q_2, \ldots, I(L_{\lambda_v}) = q_v$, and all other $I(L_\mu) \in \mathcal{A}_{r_u}^\times$. Thus, each $C_{i_u}^{r_u}$ in the input layer of the corresponding C_{i_u} multiplication unit in the second layer is activated since it has a threshold of $l - 0.5$ and receives input from the l literals. The same applies to any of the $C_{i_u}^{r_w}$, but they will also get input -1 from $C_{i_u}^{r_u}$ in the output layer of the multiplication unit. Equally, $C_{i_u}^{r_u}$ in the output layer will receive no negative input since no $C_{i_u}^{r}$ with $r <_\times r_u$ can be activated in the input layer, and thus only $C_{i_u}^{r_u}$ will be activated in the output layer of the multiplication unit. Next, A^s in the input layer of the A addition unit in the third layer receives positive input from the h clauses of which A is the head. All A^t with $s <_+ t$ will also be activated in the input layer of the A addition

unit, but not in the output layer of this unit, since A^t in the output layer will receive an input of -1 from A^s in the input layer. However, A^s will be activated, as required, because it receives an input from A^s in the input layer and no negative input because none of the A^t with $t <_+ s$ can be activated.

Conversely, if A^s is activated in the output layer of its addition unit in the third layer, then no A^t with $t <_+ s$ can be activated, by construction of an addition unit. Hence, the smallest t relative to $\leq_+$ such that all the activated $C_i^{t_j}$ have $t_j \in \mathcal{A}_t^+$ must be equal to s. Thus, the sum $\sum_{i=1}^{h} I(C_i) = s$, and accordingly there must exist a j with $R_s^{+,j} = \{r_1, r_2, \ldots, r_v\} \subseteq \{I(C_i) \mid 1 \leq i \leq h\}$. For any clause with C^r activated, where $r \in \mathcal{A}_s^+$, r is the smallest value relative to $\leq_+$ which is activated in the C multiplication unit in the second layer. There cannot be any C^t activated with $t <_\times r$ otherwise the body $L_1, L_2, \ldots, L_l$ would evaluate to t under the present interpretation I, and not to r as asserted. Thus, for this clause, we have that $\{I(L_i) \mid 1 \leq i \leq l\} \subseteq \mathcal{A}_r^\times$ and r is the smallest value relative to $\leq_\times$ for which this is the case. Therefore, $I(L_1) \times \ldots \times I(L_l) = r$, as desired.

Thus, $\mathfrak{T}_{P,\mathfrak{C}}(I)(A) = s$ if and only if A^s is activated in the third layer, and the proof is complete.

Notice that, as a corollary of this result, for each propositional logic program P, one can construct networks as given in Theorem 2, which compute, respectively, not only the classical consequence operator T_P, but also the three-valued operator Φ_P of Fitting, and the four-valued operator Ψ_P of Fitting. Indeed, as shown in [16], one can construct conventional 3-layer feedforward networks to compute Φ_P and Ψ_P containing only binary threshold units and not using multiplication and addition units. [6]

Example 8. Take $\mathfrak{C}$ as $\mathcal{FOUR}$ again and consider the program P whose clauses are $A \leftarrow A, b$; $A \leftarrow D, \neg E$; $A \leftarrow u$; $D \leftarrow$.

Here, $B_P = \{A, D, E\}$, and P^* contains the five clauses: $A \leftarrow A, b$; $A \leftarrow D, \neg E$; $A \leftarrow u$; $D \leftarrow t$; $E \leftarrow f$, which we list as 1 to 5 as shown. Also, P^{**} contains the three clauses: $A \leftarrow (A, b) + (D, \neg E) + u$; $D \leftarrow t$; $E \leftarrow f$. Thus, for any interpretation $I : B_P \rightarrow \mathcal{FOUR}$, we have $\mathfrak{T}_{P,\mathfrak{C}}(I)(A) = (I(A) \wedge b) \vee (I(D) \wedge \neg I(E)) \vee u$, $\mathfrak{T}_{P,\mathfrak{C}}(I)(D) = I(t) = t$, and $\mathfrak{T}_{P,\mathfrak{C}}(I)(E) = I(f) = f$.

For the network $\mathcal{F}$ produced by applying Theorem 2, we have $m = 3$, $n = 4$ and $K = 5$. Thus, the first layer contains three vectors (each of length 4) representing A, D and E, and the third layer contains three addition units corresponding to A, D, E with $k = 3$, 1 and 1 respectively. The middle layer contains five multiplication units corresponding to the five elements of P^*. The first of these has $l = 2$, has a factor fixed at b and is connected to the unit A in the first layer. The second multiplication unit has $l = 2$ and is

[6] See the thesis of Yvonne Kalinke: "Ein massiv paralleles Berechnungsmodell für normale logische Programme", Department of Computer Science, Dresden University of Technology, 1994, where these results are stated. We thank S. Hölldobler for drawing this reference to our attention.

connected to units D and E in the first layer. The other three units in the second layer are $\mathfrak{C}$-element units for u, t and f respectively, and each may be connected to E, say, in the first layer. On giving an interpretation I to $\mathcal{F}$ as input, it is readily checked that $\mathcal{F}$ computes $\mathfrak{T}_{P,\mathfrak{C}}(I)$.

11.4 Annoted Logic Programs and the Operator $\mathcal{T}_P$

11.4.1 Lattice and Bilattice-Based Annotated Logic Programs

We now turn to the second class of extended logic programs we wish to consider: annotated (bi)lattice-based logic programs. For a detailed exposition of lattice- and bilattice-based annotated logic programs, see [35, 15]. The so-called generalized annotated logic programs (GAPs) were introduced in [35] and were shown to generalize annotation-free, and implication-based logic programs as well as most of the various annotated logic programs introduced to date. In [15], GAPs were extended to bilattice-based annotated logic programs (BAPs), which allowed us to introduce a continuous semantic operator in place of the non-continuous semantic operator of [35]. We will therefore take BAPs as the most general approach to annotated logic programming, and use the semantic operator introduced for them, but we will concentrate, in the main, only on a one-lattice fragment of BAPs in order to bring uniformity into the discussion of the current section and the previous section which analysed sets of truth values with only one ordering defined on them. This permits us to claim that the results described in the previous section are extendable to BAPs with only variables allowed in annotations.

The notion of a bilattice was introduced in the 1980s as a generalization of the famous lattice $\mathcal{FOUR}$ of Belnap (see Example 1) as a suitable structure for interpreting different languages and programs when working with uncertainty and incomplete or inconsistent databases, see [28, 29, 21, 22, 35] for further details and further motivation.

Definition 7. *[21] A* bilattice $\mathbf{B}$ *is a sextuple* $(\mathbf{B}, \vee, \wedge, \oplus, \otimes, \neg)$ *such that* $(\mathbf{B}, \vee, \wedge)$ *and* $(\mathbf{B}, \oplus, \otimes)$ *are both complete lattices, and* $\neg : \mathbf{B} \to \mathbf{B}$ *is a mapping satisfying the following three properties:* $(\neg)^2 = Id_{\mathbf{B}}$, $\neg$ *is a dual lattice homomorphism from* $(\mathbf{B}, \vee, \wedge)$ *to* $(\mathbf{B}, \wedge, \vee)$, *and* $\neg$ *is a lattice homomorphism from* $(\mathbf{B}, \oplus, \otimes)$ *to itself.*

The lattice $(\mathbf{B}, \vee, \wedge)$ is traditionally thought of as generalizing the Boolean lattice $\{\text{false}, \text{true}\}$, and is used for describing measures of truth and falsity. The lattice $(\mathbf{B}, \oplus, \otimes)$ is thought of as measuring the amount of information (or knowledge) between none and both (as in $\mathcal{FOUR}$).

Let $(L_1, \leq_1)$ and $(L_2, \leq_2)$ denote two lattices, let x_1, x_2 denote arbitrary elements of the lattice L_1, and let y_1, y_2 denote arbitrary elements of the lattice L_2. Let $\cap_1, \cup_1$ denote the meet respectively join defined in the lattice L_1, and let $\cap_2, \cup_2$ denote the meet respectively join defined in the lattice L_2.

Now form the set of points $L_1 \times L_2$. We define the usual orderings $\leq_t$ (the truth ordering) and $\leq_k$ (the knowledge ordering) on $L_1 \times L_2$ as follows.

(1) $\langle x_1, y_1\rangle \leq_t \langle x_2, y_2\rangle$ if and only if $x_1 \leq_1 x_2$ and $y_2 \leq_2 y_1$.
(2) $\langle x_1, y_1\rangle \leq_k \langle x_2, y_2\rangle$ if and only if $x_1 \leq_1 x_2$ and $y_1 \leq_2 y_2$.

We use here the fact that each distributive bilattice can be regarded as a product of two lattices, see [29]. For convenience of presentation, we will treat each bilattice we work with as isomorphic to some subset of $\mathbf{B} = L_1 \times L_2 = ([0,1], \leq) \times ([0,1], \leq)$, where $[0,1]$ is the unit interval of real numbers with the linear ordering defined on it. Elements of such a bilattice are pairs. In particular, $(1,0)$ and $(0,1)$ are the analogues of true and false and are maximal respectively minimal in the truth ordering, whilst $(1,1)$ (or both) and $(0,0)$ (or none) are respectively maximal and minimal elements in the knowledge ordering.

We define an annotated bilattice-based language $\mathcal{L}$ to consist of individual variables, constants, functions and predicate symbols together with annotation terms which can consist of variables, constants and/or functions over a bilattice. Bilattice-based languages allow, in general, six connectives and four quantifiers, as follows: $\oplus, \otimes, \vee, \wedge, \neg, \sim, \mathit{\Sigma}, \mathit{\Pi}, \exists, \forall$. But in this chapter we restrict our attention to only one-lattice based BAPs and will work only with $\oplus, \otimes, \mathit{\Sigma}$, the latter being the existential quantifier with respect to the knowledge ordering. Returning to algebraic characterizations of many-valued logics, we make the remark that $\oplus$ and $\otimes$ correspond to the operations $+$ and $\times$ of Section 11.3.1, and that $\mathit{\Sigma}$ corresponds to infinite summation $(+)$.

An *annotated formula* is defined inductively as follows: if R is an n-ary predicate symbol, $t_1, \ldots, t_n$ are terms, and μ is an annotation term, then $R(t_1, \ldots, t_n) : \mu$ is an *annotated formula* (called an *annotated atom*). Annotated atoms can be combined to form complex formulae using the connectives and quantifiers.

A *bilattice-based annotated logic program (BAP)* P consists of a finite set of *annotated program clauses* of the form

$$A : \mu \leftarrow L_1 : \mu_1, \ldots, L_n : \mu_n,$$

where $A : \mu$ denotes an annotated atom called the *head* of the clause, and $L_1 : \mu_1, \ldots, L_n : \mu_n$ denotes $L_1 : \mu_1 \otimes \ldots \otimes L_n : \mu_n$ and is called the *body* of the clause; each $L_i : \mu_i$ is an annotated literal called an *annotated body literal* of the clause. Individual and annotation variables in the body are thought of as being existentially quantified using $\mathit{\Sigma}$. In [15], we showed how the remaining connectives $\oplus, \vee, \wedge$ can be introduced into BAPs, but we will not address this issue here.

Each annotated atom $A : \mu$ is interpreted in two steps as follows: the first-order atomic formula A is interpreted in $\mathbf{B}$ (we may write $\mathcal{I}_{\mathbf{B}}(A) \rightarrow \mathbf{B}$ to indicate this process) using a domain of interpretation and a variable assignment, see [28, 29, 35, 15] for further details. Then we define the interpretation

$I_{\mathbf{B}}$ as follows: if $\mathcal{I}_{\mathbf{B}}(A) \geq \mu$, we put $I_{\mathbf{B}}(A : \mu) = 1$, and $I_{\mathbf{B}}(A : \mu) = 0$ otherwise.

Let $I_{\mathbf{B}}$ be an interpretation for $\mathcal{L}$ and let F be a closed annotated formula of $\mathcal{L}$. Then $I_{\mathbf{B}}$ is a *model* for F if $I_{\mathbf{B}}(F) = 1$. We say that $I_{\mathbf{B}}$ is a model for a set S of annotated formulae if $I_{\mathbf{B}}$ is a model for each annotated formula of S. We say that F *is a logical consequence of* S if, for every interpretation $I_{\mathbf{B}}$ of $\mathcal{L}$, $I_{\mathbf{B}}$ is a model for S implies $I_{\mathbf{B}}$ is a model for F.

Let B_P and U_P denote the annotation Herbrand base and the annotation Herbrand universe for a program P respectively; they are essentially B_P and U_P as defined in [46], but with annotation terms allowed in U_P and attached to ground formulae in B_P. In common with conventional logic programming, each Herbrand interpretation HI for P can be identified with the subset $\{R(t_1, \ldots, t_k) : \alpha \in B_P \mid R(t_1, \ldots, t_k) : \alpha$ receives the value 1 with respect to $I_{\mathbf{B}}\}$ of B_P, where $R(t_1, \ldots, t_k) : \alpha$ denotes a typical element of B_P. This set constitutes an *annotation Herbrand model for* P. Finally, we let $\mathrm{HI}_{P,\mathbf{B}}$ denote the set of all annotation Herbrand interpretations for P.

It was observed in [22, 14, 15, 37, 38], that the non-linear ordering of (bi)lattices influences both model-theoretic properties and proof procedures for (bi)lattice-based logics, and this distinguishes them from classical and even fuzzy logic. In particular, both the semantic operator and SLD-resolution for BAPs must reflect the non-linear ordering of bilattices, see [13, 14, 15].

In [15], we introduced a semantic operator $\mathcal{T}_P$ for BAPs, proved its continuity and showed that it computes the least Herbrand model for a given BAP as its least fixed point.

Definition 8. *We define the mapping* $\mathcal{T}_P : \mathrm{HI}_{P,\mathbf{B}} \rightarrow \mathrm{HI}_{P,\mathbf{B}}$ *as follows:* $\mathcal{T}_P(\mathrm{HI})$ *denotes the set of all* $A : \mu \in B_P$ *such that either*

1. *There is a strictly ground instance of a clause* $A : \mu \leftarrow L_1 : \mu_1, \ldots, L_n : \mu_n$ *such that* $\{L_1 : \mu'_1, \ldots, L_n : \mu'_n\} \subseteq \mathrm{HI}$ *for some annotations* $\mu'_1, \ldots, \mu'_n$, *and one of the following conditions holds for each* μ'_i*:*
   a) $\mu'_i \geq_k \mu_i$,
   b) $\mu'_i \geq_k \otimes_{j \in J_i} \mu_j$, *where* J_i *is the finite set of those indices* $i, j \in \{1, \ldots n\}$ *such that* $L_j = L_i$

or

2. *there are annotated strictly ground atoms* $A : \mu^*_1, \ldots, A : \mu^*_k \in \mathrm{HI}$ *such that* $\mu \leq_k \mu^*_1 \oplus \ldots \oplus \mu^*_k$.[7]

Item 1a is the analogue of the conventional T_P operator, see [46] for example, and of the generalized semantic operator $\mathfrak{T}_{P,\mathfrak{C}}$. Items 1b and 2 reflect properties of the non-linear ordering defined on the set $\mathrm{HI}_{P,\mathbf{B}}$ of all interpretations, as further illustrated in the next example. Note that the absence of conditions 1b and 2 in the formulation of $\mathfrak{T}_{P,\mathfrak{C}}$ given in Section 11.3 is

[7] Note that whenever $F : \mu \in HI$ and $\mu' \leq_k \mu$, then $F : \mu' \in HI$. Also, for each formula F, $F : (0, 0) \in HI$.

compensated for by the use of the ground completion P^{**} of a program P whenever $\mathfrak{T}_{P,\mathfrak{C}}$ is applied.

Example 9. Consider a bilattice-based annotated logic program P which can collect and process information about connectivity of some (probabilistic) graph G. Suppose we have received information from two different sources: one reports that there is an edge between nodes a and b, the other, that there is no such. This is represented by the two unit clauses $\texttt{edge}(a,b) : (1,0) \leftarrow$, $\texttt{edge}(a,b) : (0,1) \leftarrow$. It is reasonable to conclude that the information is contradictory, that is, to conclude that $\texttt{edge}(a,b) : (1,1)$, and this fact is captured by item 2. If, on the other hand, the program contains some clause of the form $\texttt{disconnected}(G) : (1,1) \leftarrow \texttt{connected}(a,c) : (1,0), \texttt{connected}(a,c) : (0,1)$, we may regard the clause $\texttt{disconnected}(G) : (1,1) \leftarrow \texttt{connected}(a,c) : (0,0)$ as equal to the initial clause. This fact is captured by item 1b.

Let B, A and C denote, respectively, $\texttt{edge}(a,b)$, $\texttt{connected}(a,c)$ and $\texttt{disconnected}(G)$. Consider the logic program: $B : (0,1) \leftarrow$, $B : (1,0) \leftarrow$, $A : (0,0) \leftarrow B : (1,1), C : (1,1) \leftarrow A : (1,0), A : (0,1)$. The least fixed point of $\mathcal{T}_P$ is $\mathcal{T}_P \uparrow 3 = \{B : (0,1), B : (1,0), B : (1,1), A : (0,0), C : (1,1)\}$. However, the item 1.$a$ (corresponding to the classical semantic operator) would allow us to compute only $\mathcal{T}_P \uparrow 1 = \{B : (0,1), B : (1,0)\}$, that is, to compute only explicit consequences of the program, which then leads to a contradiction in the two-valued case.

As was shown in [15], the BAPs introduced here generalize different sorts of implication-based and annotation-free logic programs, such as those of [28, 29]. The fragment of BAPs described here is computationally equivalent to GAPs - annotated logic programs based on one-lattice structures. This gives a very close connection between the structures used to interpret GAPs respectively BAPs and captured by the algebraic analysis of the operations $+$ and $\times$ on the set $\mathfrak{C}$, as described in Section 11.3.1. Namely, both GAPs and the fragment of BAPs described here, both taken only with variable annotations, yield the general semantic characterizations of Sections 11.3.1 and 11.3.2, as follows.

Definition 9. *Let P be an annotation-free logic program with clauses of the form $A \leftarrow B_1, \ldots, B_n$. We construct P',* an annotated logic program derived from P, *as follows. For each clause $A \leftarrow B_1, \ldots, B_n$ in P, add a clause $A : \mu \leftarrow B_1 : \mu_1, \ldots, B_n : \mu_n$ to P', where each μ_i is an annotation variable.*

The following proposition is an adaptation of the relevant proposition from [35, 15].

Proposition 4. *Let A be a ground first-order atomic formula and let α be an annotation constant. Then $I(A) = \alpha$ in the least fixed point of $\mathfrak{T}_{P,\mathfrak{C}}$ if and only if $A : \alpha$ is in the least fixed point of $\mathcal{T}_P$.*

We now turn to the description of the neural networks computing $\mathcal{T}_P$.

11.4.2 Neural Networks for (Bi)Lattice-Based Annotated Logic Programs

We extend the approach of [1] described in Section 11.3 to learning neural networks which can compute logical consequences of BAPs. This will allow us to introduce hypothetical and uncertain reasoning based on BAPs into the framework of neural-symbolic computation. Bilattice-based logic programs can work with conflicting sources of information and inconsistent databases. Therefore, neural networks corresponding to these logic programs should reflect this facility as well, and this is why we introduce some forms of learning into neural networks. These forms of learning can be seen as corresponding to unsupervised Hebbian learning which is widely implemented in neurocomputing. The general idea behind Hebbian learning is that positively correlated activities of two neurons strengthen the weight of the connection between them and uncorrelated or negatively correlated activities weaken the weight of the connection between them (the latter form is known as anti-Hebbian learning).

The general conventional definition of Hebbian learning is given as follows, see [51] for further details. Let k and j denote two units and w_{kj} denote the weight of a connection from j to k. We denote the value of j at time t by $v_j(t)$ and the potential of k at time t by $p_k(t)$. Then the rate of change in the weight between j and k is expressed in the form

$$\Delta w_{kj}(t) = F(v_j(t), p_k(t)),$$

where F is some function. As a special case of this formula, it is common to write

$$\Delta w_{kj}(t) = \eta(v_j(t))(p_k(t)),$$

where η is a constant that determines the *rate of learning* and is positive in the case of Hebbian learning and negative in the case of anti-Hebbian learning. Finally, we update by $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$.

In this section, we will compare the two learning functions we introduce with this conventional definition of Hebbian learning. First, we prove a theorem establishing a relationship between learning neural networks and BAPs with no function symbols occurring in either individual or annotation terms. (Since the annotation Herbrand base for these programs is finite, they can equivalently be seen as propositional bilattice-based logic programs with no functions allowed in the annotations.)

Theorem 3. *For each function-free BAP P, there exists a 3-layer feedforward learning neural network which computes $\mathcal{T}_P$.*

Proof. Let m and n be the number of strictly ground annotated atoms from the annotation Herbrand base B_P and the number of clauses occurring in P respectively. Without loss of generality, we may assume that the annotated atoms are ordered. The network associated with P can now be constructed by the following translation algorithm.

1. The input and output layers are vectors of binary threshold units of length k, $1 \leq k \leq m$, where the i-th unit in the input and output layers represents the i-th strictly ground annotated atom. The threshold of each unit occurring in the input or output layer is set to 0.5.
2. For each clause of the form $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_m : (\alpha_m, \beta_m)$, $m \geq 0$, in P, do the following.
   2.1 Add a binary threshold unit c to the hidden layer.
   2.2 Connect c to the unit representing $A : (\alpha, \beta)$ in the output layer with weight 1. We will call connections of this type 1-*connections*.
   2.3 For each atom $B_j : (\alpha_j, \beta_j)$ in the input layer, connect the unit representing $B_j : (\alpha_j, \beta_j)$ to c and set the weight to 1. (We will call these connections 1-connections also.)
   2.4 Set the threshold θ_c of c to $l - 0.5$, where l is the number of atoms in $B_1 : (\alpha_1, \beta_1), \ldots, B_m : (\alpha_m, \beta_m)$.
   2.5 If some input unit representing $B : (\alpha, \beta)$ is connected to a hidden unit c, connect each of the input units representing annotated atoms $B : (\alpha_i, \beta_i), \ldots, B : (\alpha_j, \beta_j)$ to c. These connections will be called $\otimes$-*connections*. The weights of these connections will depend on a learning function. If the function is inactive, set the weight of each $\otimes$-*connection* to 0.
3. If there are units representing atoms of the form $B : (\alpha_i, \beta_i), \ldots, B : (\alpha_j, \beta_j)$ in the input and output layers, correlate them as follows. For each $B : (\alpha_i, \beta_i)$, connect the unit representing $B : (\alpha_i, \beta_i)$ in the input layer to each of the units representing $B : (\alpha_i, \beta_i), \ldots, B : (\alpha_j, \beta_j)$ in the output layer. These connections will be called $\oplus$-*connections*. If an $\oplus$-connection is set between two atoms with different annotations, we consider them as being connected via hidden units with thresholds 0. If an $\oplus$-connection is set between input and output units representing the same annotated atom $B : (\alpha, \beta)$, we set the threshold of the hidden unit connecting them to -0.5, and we will call them $\oplus$-hidden units, so as to distinguish the hidden units of this type. The weights of all these $\oplus$-connections will depend on a learning function. If the function is inactive, set the weight of each $\oplus$-*connection* to 0.
4. Set all the weights which are not covered by these rules to 0. For each annotated atom $A : (\alpha, \beta)$, connect the unit representing $A : (\alpha, \beta)$ in the output layer to the unit representing it in the input layer with weight 1.

Allow two learning functions to be embedded into the $\otimes$ -connections and the $\oplus$ -connections. We let v_i denote the value of the input unit representing $B : (\alpha_i, \beta_i)$ and let p_c denote the potential of the unit c.

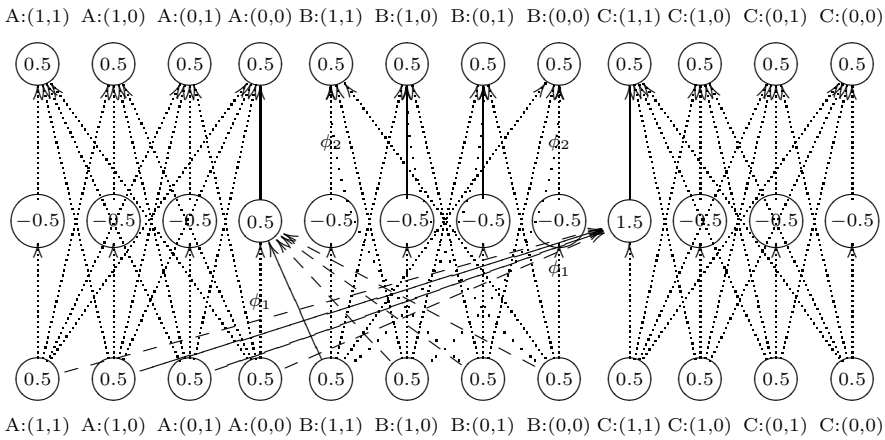
Let a unit representing $B : (\alpha_i, \beta_i)$ in the input layer be denoted by i. If i is connected to a hidden unit c via an $\otimes$ -connection, then a learning function ϕ_1 is associated to this connection as defined next. We let $\phi_1 = \Delta w_{ci}(t-1) = v_i(t-1)(-p_c(t-1)+0.5)$ become active and change the weight of the $\otimes$-connection from i to c at time t if i became activated at time $(t-1)$;

units representing atoms $B : (\alpha_j, \beta_j), \ldots, B : (\alpha_k, \beta_k)$ in the input layer are connected to c via 1-connections, and $(\alpha_i, \beta_i) \geq_k (\alpha_j, \beta_j) \otimes \ldots \otimes (\alpha_k, \beta_k)$.

Function ϕ_2 is embedded only into connections of type $\oplus$, namely, into $\oplus$-connections between hidden and output layers. Let o be an output unit representing an annotated atom $B : (\alpha_i, \beta_i)$. Apply $\phi_2 = \Delta w_{oc}(t-2) = v_c(t-2)(p_o(t-2)+1.5)$ to change w_{oc} at time t if (i) ϕ_2 is embedded into an $\oplus$-connection from the $\oplus$-hidden unit c to o, and there are output units representing annotated atoms $B : (\alpha_j, \beta_j), \ldots, B : (\alpha_k, \beta_k)$ which are connected to the unit o via $\oplus$-connections, and (ii) these output units became activated at time $t-2$ and $(\alpha_i, \beta_i) \leq_k (\alpha_j, \beta_j) \oplus \ldots \oplus (\alpha_k, \beta_k)$.

Each annotation Herbrand interpretation HI for P can be represented by a binary vector $(v_1, \ldots, v_m)$. Such an interpretation is given as input to the network by externally activating corresponding units of the input layer at time t_0. It remains to show that $A : (\alpha, \beta) \in \mathcal{T}_P \uparrow n$ for some n if and only if the unit representing $A : (\alpha, \beta)$ becomes active at time $t+2$, for some t. The proof that this is so proceeds by routine induction.

Example 10. The following diagram displays the neural network which computes $\mathcal{T}_P \uparrow 3$ from Example 9. Without the functions ϕ_1, ϕ_2, the neural network will compute only $\mathcal{T}_P \uparrow 1 = \{B : (0,1), B : (1,0)\}$, and these are explicit logical consequences of the program. Indeed, it is the use of ϕ_1 and ϕ_2 that allows the neural network to compute $\mathcal{T}_P \uparrow 3$. Note that the arrows $\longrightarrow$, $- - \succ$, $\cdots\succ$ denote respectively 1-connections, $\otimes$-connections and $\oplus$-connections, and we have marked by ϕ_1, ϕ_2 the connections which are activated by the learning functions.[8]



We can make several conclusions from the construction of Theorem 3.

[8] We do not draw here the connections which make this network recurrent.

- Neurons representing annotated atoms with identical first-order (or propositional) components are joined into multineurons in which units are correlated using $\oplus$- and $\otimes$-connections.
- The learning function ϕ_2 roughly corresponds to Hebbian learning, with the rate of learning $\eta_2 = 1$, the learning function ϕ_1 corresponds to anti-Hebbian learning with the rate of learning $\eta_1 = -1$, and we regard η_1 as negative because the factor p_c in the formula for ϕ_1 is multiplied by (-1).
- The main problem Hebbian learning causes is that the weights of connections with embedded learning functions tend to grow exponentially, which cannot fit the model of biological neurons. This is why traditionally functions are introduced to bound the growth. In the neural networks we have built, some of the weights may grow with iterations, but the growth will be very slow because we are using binary threshold units in the computation of each v_i.

11.5 Relationships Between the Neural Networks Simulating $\mathfrak{T}_{P,\mathfrak{C}}$ and $\mathcal{T}_P$

In this section, we will briefly compare the neural networks computing $\mathfrak{T}_{P,\mathfrak{C}}$ and $\mathcal{T}_P$, and establish a result which relates the computations performed by these two neural networks.

Both the ANNs constructed here can be seen as direct extensions of the work of [1] in that the fundamental processing unit in each case is the binary threshold unit, including those units employed in multiplication and adddition units. This property of the neural networks of Section 11.4 is a direct consequence of the fact that annotated logic programs eventually receive a two-valued meaning. The neural networks of Section 11.3 could be designed to process many-valued vectors, as they are built upon annotation-free logic programs which receive their meaning only via many-valued interpretations, but we use only binary vectors here.

The main difference between the two types of neural networks is that the ANNs of Section 11.3 require two additional layers in comparison with those of [1] and Section 11.4, although their connections are less complex. On the other hand, the ANNs of Section 11.4 have complex connections and compensate for "missing" layers by applying learning functions in the spirit of neurocomputing. The latter property can perhaps be seen as an optimization of the ANNs built in Section 11.3, although the ANNs of Section 11.3 can compute many-valued semantic operators without using learning functions.

We close the chapter with the following important result.

Theorem 4. *Let P denote an annotation-free logic program, and let P' denote the annotated logic program derived from P. Then there exist an ANN1 (as constructed in Section 11.3.4) simulating $\mathfrak{T}_{P,\mathfrak{C}}$ and an ANN2 (as constructed in Section 11.4.2) simulating $\mathcal{T}_{P'}$ such that the output vector of ANN1 at each time $t+2$ is equal to the output vector of ANN2 at time t.*

Proof. We use Definition 9 and Proposition 4.

Without loss of generality, we assume that all the ground atoms and all the values are ordered. When constructing the networks ANN1 and ANN2, we assume also that all the input and output units of ANN1 and ANN2 are synchronized in that the order at which each m-th proposition with n-th value appears in ANN1 corresponds to the order in which each unit representing each $A_m : \mu_n$ appears in ANN2.

The rest of the proof makes use of the constructions of Theorems 2 and 3 and proceeds by routine induction on the number of iterations of $\mathfrak{T}_{P,\mathfrak{C}}$ and $\mathcal{T}_P$.

11.6 Conclusions and Further Work

We have given a very general algebraic characterization of many-valued annotation-free logic programs, and have shown how this analysis can be extended to other types of many-valued logic programs. We have also shown how two very general semantic operators, the generalized semantic operator $\mathfrak{T}_{P,\mathfrak{C}}$ for annotation-free logic programs and the enriched semantic operator $\mathcal{T}_P$ for annotated logic programs, can be defined, and we have established semantical and computational relationships between them.

Furthermore, we have proposed neural networks in the style of [1] for computing $\mathfrak{T}_{P,\mathfrak{C}}$ and $\mathcal{T}_P$. The neural networks we have given for computing $\mathfrak{T}_{P,\mathfrak{C}}$ require several additional layers in order to reflect the many-valued properties of $\mathfrak{T}_{P,\mathfrak{C}}$ that they simulate. On the other hand, the neural networks computing $\mathcal{T}_P$ have learning functions embedded in them which compensate for the use of additional layers. It would be interesting to carry out a detailed analysis of the complexity (time and space) of both of these neural networks and to compare them on complexity grounds.

Future work in the general direction of the chapter includes the following.

1. Further analysis of the properties of many-valued semantic operators, as given, for example, in [31], and its implications for the construction of the corresponding networks.
2. The neural networks computing the operators considered here could perhaps be optimized if transformed into non-binary neural networks. This might result, for example, in the removal of the annotations used in the representation of input and output units.
3. Another area of our future research is to investigate how learning can optimize and improve the representation of different neuro-symbolic systems and the computations performed by them.

References

1. Hölldobler, S., Kalinke, Y.: Towards a new massively parallel computational model for logic programming. In: Proceedings ECAI94 Workshop on Combining Symbolic and Connectionist Processing, ECCAI (1994) 68–77
2. Hölldobler, S., Kalinke, Y., Störr, H.P.: Approximating the semantics of logic programs by recurrent neural networks. Applied Intelligence **11** (1999) 45–58
3. Bader, S., Hitzler, P., Hölldobler, S.: The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. Information **9**(1) (2006) Invited paper.
4. Bornscheuer, S.E., Hölldobler, S., Kalinke, Y., Strohmaier, A.: Massively parallel reasoning. In: Automated Deduction – A Basis for Applications. Volume II. Kluwer Academic Publishers (1998) 291–321
5. d'Avila Garcez, A.S., Broda, K., Gabbay, D.M.: Symbolic knowledge extraction from trained neural networks: A sound approach. Artificial Intelligence **125** (2001) 155–207
6. d'Avila Garcez, A.S., Broda, K.B., Gabbay, D.M.: Neural-Symbolic Learning Systems — Foundations and Applications. Perspectives in Neural Computing. Springer, Berlin (2002)
7. d'Avila Garcez, A.S., Gabbay, D.: Fibring neural networks. In McGuinness, D., Ferguson, G., eds.: Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence, July 2004, San Jose, California, USA, AAAI Press/The MIT Press (2004) 342–347
8. d'Avila Garcez, A.S., Lamb, L.C., Gabbay, D.M.: A connectionist inductive learning system for modal logic programming. In: Proceedings of the IEEE International Conference on Neural Information Processing ICONIP'02, Singapore. (2002)
9. Fu, L.: Neural Networks in Computer Intelligence. McGraw-Hill, Inc. (1994)
10. Hitzler, P., Hölldobler, S., Seda, A.K.: Logic programs and connectionist networks. Journal of Applied Logic **2**(3) (2004) 245–272
11. Hitzler, P., Seda, A.K.: Continuity of semantic operators in logic programming and their approximation by artificial neural networks. In Günter, A., Krause, R., Neumann, B., eds.: Proceedings of the 26th German Conference on Artificial Intelligence, KI2003. Volume 2821 of Lecture Notes in Artificial Intelligence., Springer (2003) 105–119
12. Hölldobler, S.: Challenge problems for the integration of logic and connectionist systems. In Bry, F., Geske, U., Seipel, D., eds.: Proceedings 14. Workshop Logische Programmierung. Volume 90 of GMD Report., GMD (2000) 161–171
13. Komendantskaya, E., Seda, A.K.: Declarative and operational semantics for bilattice-based annotated logic programs. In Li, L., Ren, F., Hurley, T., Komendantsky, V., an Airchinnigh, M.M., Schellekens, M., Seda, A., Strong, G., Woods, D., eds.: Proceedings of the Fourth International Conference on Information and the Fourth Irish Conference on the Mathematical Foundations of Computer Science and Information Technology, NUI, Cork, Ireland. (2006) 229–232
14. Komendantskaya, E., Seda, A.K.: Logic programs with uncertainty: neural computations and automated reasoning. In: Proceedings of the International Conference Computability in Europe (CiE), Swansea, Wales (2006) 170–182

15. Komendantskaya, E., Seda, A.K., Komendantsky, V.: On approximation of the semantic operators determined by bilattice-based logic programs. In: Proc. 7th International Workshop on First-Order Theorem Proving (FTP'05), Koblenz, Germany (2005) 112–130
16. Lane, M., Seda, A.K.: Some aspects of the integration of connectionist and logic-based systems. Information **9**(4) (2006) 551–562
17. Seda, A.K.: Morphisms of neural networks. Technical report, Department of Mathematics, University College Cork, Ireland (2006)
18. Seda, A.K.: On the integration of connectionist and logic-based systems. In Hurley, T., an Airchinnigh, M.M., Schellekens, M., Seda, A.K., Strong, G., eds.: Proceedings of MFCSIT'04, Trinity College Dublin, July, 2004. Volume 161 of Electronic Notes in Theoretical Computer Science (ENTCS)., Elsevier (2006) 109–130
19. Bader, S., Hitzler, P., Hölldobler, S., Witzel, A.: A fully connectionist model generator for covered first-order logic programs. In: M. Veloso, ed.: Proceedings of International Joint Conference on Artificial Intelligence IJCAI07, Hyderabad, India, AAAI Press (2007) 666–671
20. Resher, N.: Many-valued logic. Mac Graw Hill (1996)
21. Ginsberg, M.L.: Multivalued logics: A uniform approach to inference in artificial intelligence. Computational Intelligence **4**(3) (1992) 256–316
22. Kifer, M., Lozinskii, E.L.: RI: A logic for reasoning with inconsistency. In: Proceedings of the 4th IEEE Symposium on Logic in Computer Science (LICS), Asilomar, IEEE Computer Press (1989) 253–262
23. Lu, J.J.: Logic programming with signs and annotations. Journal of Logic and Computation **6**(6) (1996) 755–778
24. Fitting, M.: A Kripke-Kleene semantics for general logic programs. The Journal of Logic Programming **2** (1985) 295–312
25. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint logic programming: Syntax and semantics. ACM Transactions on Programming Languages and Systems (TOPLAS) **23**(1) (2001) 1–29
26. Sessa, M.I.: Approximate reasoning by similarity-based SLD-resolution. Theoretical computer science **275** (2002) 389–426
27. Vojtás, P., Paulik, L.: Soundness and completeness of non-classical extended sld-resolution. In: Extensions of Logic Programming, 5th International Workshop ELP'96, Leipzig, Germany, March 28-30, 1996. Volume 1050 of Lecture notes in Computer Science., Springer (1996) 289–301
28. Fitting, M.: Bilattices and the semantics of logic programming. The Journal of Logic Programming **11** (1991) 91–116
29. Fitting, M.C.: Bilattices in logic programming. In Epstein, G., ed.: The twentieth International Symposium on Multiple-Valued Logic, IEEE (1990) 238–246
30. Fitting, M.C.: Kleene's logic, generalized. Journal of Logic and Computation **1** (1992) 797–810
31. Damásio, C.V., Pereira, L.M.: Sorted monotonic logic programs and their embeddings. In: Proceedings of the 10th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU-04). (2004) 807–814
32. Stamate, D.: Quantitative datalog semantics for databases with uncertain information. In Pierro, A.D., Wiklicky, H., eds.: Proceedings of the 4th Workshop on Quantitative Aspects of Programming Languages (QAPL 2006). Electronic

Notes in Theoretical Computer Science, Vienna, Austria, Elsevier (2006) To appear.
33. van Emden, M.: Quantitative deduction and its fixpoint theory. Journal of Logic Programming **3** (1986) 37–53
34. Lakshmanan, L.V.S., Sadri, F.: On a theory of probabilistic deductive databases. Theory and Practice of Logic Programming **1**(1) (2001) 5–42
35. Kifer, M., Subrahmanian, V.S.: Theory of generalized annotated logic programming and its applications. Journal of logic programming **12** (1991) 335–367
36. Calmet, J., Lu, J.J., Rodriguez, M., Schü, J.: Signed-formula logic programming: Operational semantics and applications. In: In Proceedings of the Ninth International Symposium on Foundations of Intelligent Systems. Volume 1079 of Lecture Notes in Artificial Intelligence., Berlin, Springer (1996) 202–211
37. Lu, J.J., Murray, N.V., Rosental, E.: A framework for automated reasoning in multiple-valued logics. Journal of Automated Reasoning **21**(1) (1998) 39–67
38. Lu, J.J., Murray, N.V., Rosental, E.: Deduction and search strategies for regular multiple-valued logics. Journal of Multiple-valued logic and soft computing **11** (2005) 375–406
39. Ng, R.: Reasoning with uncertainty in deductive databases and logic programs. International Journal of Uncertainty, Fuzziness and Knowledge-based Systems **2**(3) (1997) 261–316
40. Straccia, U.: Query answering in normal logic programs under uncertainty. In: 8th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-05). Lecture Notes in Computer Science, Barcelona, Spain, Springer Verlag (2005) 687–700
41. Fitting, M.: Fixpoint semantics for logic programming — A survey. Theoretical Computer Science **278**(1–2) (2002) 25–51
42. Hecht-Nielsen, R.: Neurocomputing. Addison-Wesley (1990)
43. Hertz, J., Krogh, A., Palmer, R.: Introduction to the Theory of Neural Computation. Addison-Wesley Publishing Company (1991)
44. Seda, A.K., Lane, M.: On the measurability of the semantic operators determined by logic programs. Information **8**(1) (2005) 33–52
45. Hitzler, P., Seda, A.K.: Characterizations of classes of programs by three-valued operators. In Gelfond, M., Leone, N., Pfeifer, G., eds.: Logic Programming and Non-monotonic Reasoning, Proceedings of the 5th International Conference on Logic Programming and Non-Monotonic Reasoning, LPNMR'99, El Paso, Texas, USA. Volume 1730 of Lecture Notes in Artificial Intelligence., Springer, Berlin (1999) 357–371
46. Lloyd, J.W.: Foundations of Logic Programming. Springer, Berlin (1987)
47. Wendt, M.: Unfolding the well-founded semantics. Journal of Electrical Engineering, Slovak Academy of Sciences **53**(12/s) (2002) 56–59
48. Dung, P.M., Kanchanasut, K.: A fixpoint approach to declarative semantics of logic programs. In Lusk, E.L., Overbeek, R.A., eds.: Logic Programming, Proceedings of the North American Conference 1989, NACLP'89, Cleveland, Ohio, MIT Press (1989) 604–625
49. Bonnier, S., Nilsson, U., Näslund, T.: A simple fixed point characterization of the three-valued stable model semantics. Information Processing Letters **40**(2) (1991) 73–78
50. Lane, M.: $\mathfrak{C}$-Normal Logic Programs and Semantic Operators: Their Simulation by Artificial Neural Networks. PhD thesis, Department of Mathematics, University College Cork, Cork, Ireland (2006)

51. Haykin, S.: Neural Networks. A Comprehensive Foundation. Macmillan College Publishing Company (1994)