Extended Theory Refinement in Knowledge-based Neural Networks

Artur S. d'Avila Garcez*
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
aag@doc.ic.ac.uk

Abstract

This paper shows that single hidden layer networks with semi-linear activation function compute the answer set semantics of extended logic programs. As a result, incomplete (nonmonotonic) theories, presented as extended logic programs, i.e. possibly containing both classical and default negation, may be refined through inductive learning in knowledge-based neural networks. Keywords: Hybrid Architectures, Extended Logic Programming, Feedforward Neural Networks.

1 Introduction

It is generally accepted that one of the main problems in building Expert Systems (which are responsible for the industrial success of Artificial Intelligence) lies in the process of knowledge acquisition, known as the "knowledge acquisition bottleneck". An alternative is the automation of this process through Machine Learning techniques[17]. Traditional machine learning techniques (either symbolic or neural) generate hypotheses by finding regularities in a set of training examples. More recently, hybrid machine learning techniques have been successfully applied by combining background knowledge (prior domain knowledge) with training examples. The idea is to benefit from better generalisation when correct background knowledge is available, and rely on the training examples to extend, or even revise, incomplete background knowledge[19]. In particular, hybrid (symbolic and neural) systems have been successfully applied in a number of areas, ranging from DNA sequence analysis [20] to software requirements specifications [6] (see [23] for a comparison).

The *Connectionist Inductive Learning and Logic Programming System* (*C-IL^{2}P*) is a hybrid system that uses a feedforward artificial neural network to integrate inductive learning from examples and background knowledge with deductive learning from logic programming[5, 7]. Starting with the background knowledge represented by a general logic program, a *Translation Algorithm* is applied generating a neural network that can be trained with examples using, for instance, *Backpropagation* [14]. The results obtained with this refined network can be explained by applying an *Extraction Algorithm*, responsible for deriving a revised logic program from the trained network. Moreover, the neural network computes the stable model of the program inserted in it or learned with examples, thus functioning as a parallel system for logic programming.

This paper formally extends the *C-IL^{2}P* system to the language of *Extended Logic Programs*, which contains classical negation ($\neg$) in addition to default negation ($\sim$). According to Lifschitz and McCarthy, commonsense knowledge can be represented more easily when *classical negation* is also available. Extended logic programs can be viewed as a fragment of Default theories (see [18]), and thus are of interest with respect to the relation between Logic Programming and nonmonotonic formalisms. The extended *C-IL^{2}P* system computes the *Answer Set Semantics* of extended logic programs[13]. As a result, it can be applied in a broader range of domains. For example, the application of *C-IL^{2}P* to power systems' fault diagnosis, described in [8] and [22], requires the use of the extended *C-IL^{2}P* with classical negation.

A number of nonmonotonic hybrid systems have been proposed recently [3, 16, 21]. However, to the best of my knowledge, no hybrid system has combined the ability to represent default negation and classical negation. By extending *C-IL^{2}P* to deal with classical negation, we are providing not only a massively parallel nonmonotonic system to compute the semantics of extended logic programming, but also a more flexible hybrid system to perform inductive learning with examples and background knowledge.

In Section 2, we recall *C-IL^{2}P's* Translation Algorithm for inserting general logic programs into neural networks. In Section 3, we extend the Translation Algorithm to al-

*This work was supported by the Brazilian Research Council CAPES and UK EPSRC grant GR/M38582 (VOICI project). I am grateful to Gerson Zaverucha and Krysia Broda for useful discussions, and to the anonymous referees for their comments.

low the insertion of extended logic programs. We also show that the network obtained computes the answer set semantics of the extended program, i.e. that the translation algorithm is correct. In Section 4 we conclude and discuss directions for future work.

2 The *C-IL²P* System

C-IL²P [5, 7] is a hybrid system that uses a *Translation Algorithm* to map general logic programs $\mathcal{P}$ (Definition 1) into single hidden layer neural networks $\mathcal{N}$ such that $\mathcal{N}$ computes the least fixpoint $T_\mathcal{P} \uparrow \omega$ of $\mathcal{P}$ (Definitions 2 and 3).

Definition 1 *(*General Logic Program*) A* general clause *is a rule of the form* $A_0 \leftarrow L_1, ..., L_n$ *, where* A_0 *is called an* atom *and* L_i $(1 \leq i \leq n)$ *is called a* literal.[1] *A negative literal* L_j *can be also written as* $\sim A_j$*, where* $\sim$ *denotes default negation. A* general logic program *is a finite set of general clauses.*[2]

Definition 2 (Immediate Consequence Operator $T_\mathcal{P}$) *Let* $\mathcal{P}$ *be a general program.* $B_\mathcal{P}$ *will denote the set of atoms occurring in* $\mathcal{P}$*, called the* Herbrand base *of* $\mathcal{P}$*. The mapping* $T_\mathcal{P} : 2^{B_\mathcal{P}} \rightarrow 2^{B_\mathcal{P}}$ *is defined as follows. Let* I *be a Herbrand* interpretation,[3] *then* $T_\mathcal{P}(I) = \{A_0 \in B_\mathcal{P} \mid A_0 \leftarrow L_1, ..., L_n$ *is a clause in* $\mathcal{P}$ *and* $\{L_1, ..., L_n\} \subseteq I\}$*, where* $\sim A_j$ *is mapped to* $false$ *(resp.* $true$*) by* I *iff* A *is mapped to* $true$ *(resp.* $false$*) by* I*.*

Definition 3 *(*Least Fixpoint $T_\mathcal{P} \uparrow \omega$ of $\mathcal{P}$*) We define* $T \uparrow \alpha = T(T \uparrow (\alpha - 1))$*, where* α *is a successor ordinal. Assuming that* $B_\mathcal{P}$ *is finite, there is some* $n \in \omega$ $(\omega = \{0, 1, 2, ...\})$ *such that* $T_\mathcal{P} \uparrow n = T_\mathcal{P} \uparrow n + 1$*. We define* $T_\mathcal{P} \uparrow \omega = T_\mathcal{P} \uparrow n$*.*

C-IL²P's Translation Algorithm provides a massively parallel model for computing the *stable model* semantics of $\mathcal{P}$, as done in [15]. Stable models were introduced in [12], using the intuition of rational beliefs from autoepistemic logic, as follows:

Definition 4 (Gelfond-Lifschitz Transformation) *Let* $\mathcal{P}$ *be a (grounded) logic program*[4]*. Given a set* I *of atoms from* $\mathcal{P}$*, let* $\mathcal{P}_I$ *be the program obtained from* $\mathcal{P}$ *by deleting: (i) each rule that has a negative literal* $\sim A$ *in its body with* $A \in I$*, and (ii) all the negative literals in the bodies of the remaining rules.*

Clearly, $\mathcal{P}_I$ is a positive program, so that it has a unique minimal Herbrand model[25]. If this model coincides with I then we say that I is a stable model of $\mathcal{P}$.

Definition 5 (Stable Models) *A Herbrand interpretation* I *of a program* $\mathcal{P}$ *is called stable iff* $T_{\mathcal{P}_I}(I) = I$*.*

[1] A literal is an atom or the negation of an atom.

[2] Throughout, we use $\sim$ for default negation and $\neg$ for classical negation.

[3] An *interpretation* is a function mapping atoms in $B_\mathcal{P}$ to $\{true, false\}$. A *model* for $\mathcal{P}$ is an interpretation that maps $\mathcal{P}$ to $true$.

[4] As usual, it is assumed that programs are ground, i.e. formed by substituting for each variable a ground term from some fixed language $\mathcal{L}$ (this process is called *instantiation*).

The intuition behind the definition of a stable model is as follows: consider a rational agent with a set of beliefs I and a set of premises $\mathcal{P}$. Then, any clause that has a literal $\sim A$ in its body when $A \in I$ is useless, and may be removed from $\mathcal{P}$. Moreover, any literal $\sim A$ with $A \notin I$ is trivial, and may be deleted from the clauses in which it appears in $\mathcal{P}$. This yields the simplified (positive) program $\mathcal{P}_I$, and if I happens to be precisely the set of atoms that follows logically from the simplified set of premises, then the set of beliefs I is stable. Hence, stable models are possible sets of belief a rational agent might hold.

In addition to being a massively parallel model for the computation of general logic programs $\mathcal{P}$, *C-IL²P* networks $\mathcal{N}$ can be trained with examples, using for instance *Backpropagation*, having $\mathcal{P}$ as background knowledge. The knowledge acquired by training could then be extracted, closing the learning cycle (as done in [24]). Knowledge extraction from trained networks is an extensive topic in its own right (see [1] for a comprehensive survey), and is out of the scope of this paper. The reader is referred to [5] for *C-IL²P's Extraction Algorithm.*

2.1 Translation Algorithm

In what follows, we recall *C-IL²P's Translation Algorithm* by presenting an example of the insertion of knowledge into the network.

Each rule (r_l) of $\mathcal{P}$ is mapped from the input layer to the output layer of $\mathcal{N}$ through one neuron (N_l) in the single hidden layer of $\mathcal{N}$. Intuitively, the *Translation Algorithm* from $\mathcal{P}$ to $\mathcal{N}$ has to implement the following conditions: *(1)* The input potential of a hidden neuron (N_l) can only exceed N_l's threshold (θ_l), activating N_l, when all the positive antecedents of r_l are assigned the truth-value $true$ while all the negative antecedents of r_l are assigned $false$; and *(2)* The input potential of an output neuron (A) can only exceed A's threshold (θ_A), activating A, when at least one hidden neuron N_l that is connected to A is activated.

Example 6 *Consider the logic program* $\mathcal{P} = \{A \leftarrow B, C, \sim D;\ A \leftarrow E, F;\ B \leftarrow \}$*. The* Translation Algorithm *derives the network* $\mathcal{N}$ *of Figure 1, setting weights* $(W's)$ *and thresholds* $(\theta's)$ *in such a way that conditions (1) and (2) above are satisfied. Note that, if* $\mathcal{N}$ *ought to be fully-connected, any other link (not shown in Figure 1) should receive weight zero initially.*

Note that, in Example 6, each input and output neuron of $\mathcal{N}$ is associated with an atom of $\mathcal{P}$. As a result, each input and output vector of $\mathcal{N}$ can be associated with an interpretation for $\mathcal{P}$. Also, each hidden neuron N_l corresponds to a rule r_l of $\mathcal{P}$. In order to compute the stable model of $\mathcal{P}$, output neuron B should feed input neuron B such that $\mathcal{N}$ is used to iterate $T_\mathcal{P}$, the fixpoint operator of $\mathcal{P}$. $\mathcal{N}$ will eventually converge to a stable state which is identical to the stable model of $\mathcal{P}$ (see [7]), whenever $\mathcal{P}$ is an *acceptable program* (Definitions 7 and 8).

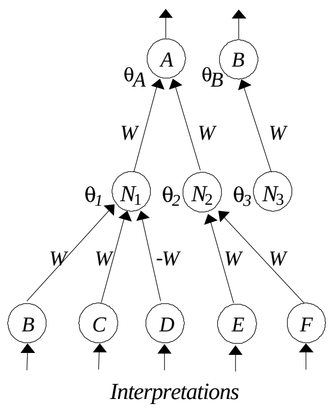

Fig. 1. Sketch of a neural network for the above program $\mathcal{P}$.

Definition 7 *A level mapping for a program $\mathcal{P}$ is a function $|\,|: B_{\mathcal{P}} \rightarrow \aleph$ of atoms to natural numbers. For $A \in B_{\mathcal{P}}$, $|A|$ is the level of A and $|\sim A| = |A|$.*

Definition 8 (Acceptable Logic Programs) *Let $\mathcal{P}$ be a general logic program, $|\,|$ a level mapping for $\mathcal{P}$, and I a model of $\mathcal{P}$. $\mathcal{P}$ is called acceptable w. r. t. $|\,|$ and I if, for every clause $A_0 \leftarrow L_1, ..., L_n$ in $\mathcal{P}$, the following implication holds for $1 \leq i \leq n$.*

$$\text{if } I \models \bigwedge_{j=1}^{i-1} L_j \text{ then } |A_0| > |L_i|$$

$\mathcal{P}$ is called acceptable if it is acceptable w. r. t. some level mapping and a model of $\mathcal{P}$.

Given a general logic program $\mathcal{P}$, let q denote the number of rules r_l $(1 \leq l \leq q)$ occurring in $\mathcal{P}$; η, the number of literals occurring in $\mathcal{P}$; A_{min}, the minimum activation for a neuron to be considered "active" (or *true*), $A_{min} \in (0,1)$; A_{max}, the maximum activation for a neuron to be considered "not active" (or *false*), $A_{max} \in (-1,0)$; $h(x) = \frac{2}{1+e^{-\beta x}} - 1$, the bipolar semi-linear activation function[5]; $g(x) = x$, the standard linear activation function; W (resp. $-W$), the weight of connections associated with positive (resp. negative) literals; θ_l, the threshold of hidden neuron N_l associated with rule r_l; θ_A, the threshold of output neuron A, where A is the head of rule r_l; k_l, the number of literals in the body of rule r_l; p_l, the number of positive literals in the body of rule r_l; n_l, the number of negative literals in the body of rule r_l; μ_l, the number of rules in $\mathcal{P}$ with the same atom in the head, for each rule r_l; $MAX_{r_l}(k_l, \mu_l)$, the greater element among k_l and μ_l for rule r_l; and $MAX_{\mathcal{P}}(k_1, ..., k_q, \mu_1, ..., \mu_q)$, the greatest element among all k's and μ's of $\mathcal{P}$. We use $\overrightarrow{k}$ as a short for $(k_1, ..., k_q)$, and $\overrightarrow{\mu}$ as a short for $(\mu_1, ..., \mu_q)$.

For instance, for program $\mathcal{P}$ (Example 6), $q = 3$, $\eta = 6$, $k_1 = 3$, $k_2 = 2$, $k_3 = 0$, $p_1 = 2$, $p_2 = 2$, $p_3 = 0$, $n_1 = 1$, $n_2 = 0$, $n_3 = 0$, $\mu_1 = 2$, $\mu_2 = 2$, $\mu_3 = 1$, $MAX_{r_1}(k_1, \mu_1) = 3$, $MAX_{r_2}(k_2, \mu_2) = 2$, $MAX_{r_3}(k_3, \mu_3) = 1$, $MAX_{\mathcal{P}}(k_1, k_2, k_3, \mu_1, \mu_2, \mu_3) = 3$.

In the *Translation Algorithm* below, we define A_{min}, W, θ_l, and θ_A such that conditions *(1)* and *(2)* above are satisfied (the proof is provided in [7]).

- **Translation Algorithm:**

Given a general logic program $\mathcal{P}$, consider that the literals of $\mathcal{P}$ are numbered from 1 to η such that the input and output layers of $\mathcal{N}$ are vectors of maximum length η, where the i-th neuron represents the i-th literal of $\mathcal{P}$. We assume, for mathematical convenience and without loss of generality, that $A_{max} = -A_{min}$.

1. Calculate $MAX_{\mathcal{P}}(\overrightarrow{k}, \overrightarrow{\mu})$ of $\mathcal{P}$;

2. Calculate the values of A_{min} and W such that the following is satisfied:

$$A_{min} > \frac{MAX_{\mathcal{P}}(\overrightarrow{k}, \overrightarrow{\mu}) - 1}{MAX_{\mathcal{P}}(\overrightarrow{k}, \overrightarrow{\mu}) + 1} \text{ and}$$

$$W \geq \frac{2}{\beta} \cdot \frac{ln\,(1 + A_{\text{min}}) - ln\,(1 - A_{\text{min}})}{MAX_{\mathcal{P}}(\overrightarrow{k}, \overrightarrow{\mu})\,(A_{\text{min}} - 1) + A_{\text{min}} + 1};$$

3. For each rule r_l of $\mathcal{P}$ of the form $L_1, ..., L_k \rightarrow A$ $(k \geq 0)$:

   (a) Add a neuron N_l to the hidden layer of $\mathcal{N}$;

   (b) Connect each neuron L_i $(1 \leq i \leq k)$ in the input layer to the neuron N_l in the hidden layer. If L_i is a positive literal then set the connection weight to W; otherwise, set the connection weight to $-W$;

   (c) Connect the neuron N_l in the hidden layer to the neuron A in the output layer and set the connection weight to W;

   (d) Define the threshold (θ_l) of the neuron N_l in the hidden layer as:

$$\theta_l = \frac{(1 + A_{\text{min}})\,(k_l - 1)}{2} W$$

   (e) Define the threshold (θ_A) of the neuron A in the output layer as:

$$\theta_A = \frac{(1 + A_{\text{min}})\,(1 - \mu_l)}{2} W$$

4. Set $g(x)$ as the activation function of the neurons in the input layer of $\mathcal{N}$. In this way, the activation of the neurons in the input layer of $\mathcal{N}$, given by each input vector **i**, will represent an interpretation for $\mathcal{P}$.

5. Set $h(x)$ as the activation function of the neurons in the hidden and output layers of $\mathcal{N}$. In this way, a gradient descent learning algorithm, such as *back-propagation*, can be applied on $\mathcal{N}$ efficiently.

[5]We use the bipolar semi-linear activation function for convenience. Any monotonically crescent activation function could have been used here.

6. If $\mathcal{N}$ ought to be fully-connected, set all other connections to zero.

Theorem 9 *[7] For each general logic program $\mathcal{P}$, there exists a feedforward artificial neural network $\mathcal{N}$ with exactly one hidden layer and semi-linear neurons such that $\mathcal{N}$ computes $T_{\mathcal{P}}$.*

In order to iterate $T_{\mathcal{P}}$, $\mathcal{N}$ is transformed into a *partially recurrent network* $\mathcal{N}_r$ by connecting each neuron in the output layer to its correspondent neuron in the input layer (e.g., B in Figure 1) with a fixed weight $W_r = 1$. In this way, the network's output vector becomes its input vector in the next computation of $T_{\mathcal{P}}$.

Theorem 10 *[9] For each acceptable general program $\mathcal{P}$, the function $T_{\mathcal{P}}$ has a unique fixpoint. The sequence of all $T_{\mathcal{P}} \uparrow m$ $(\mathbf{i}), m \in \aleph$, converges to this fixpoint $T_{\mathcal{P}} \uparrow \omega$ $(\mathbf{i})$ (which is identical to the stable model of $\mathcal{P}$), for each $\mathbf{i} \subseteq B_{\mathcal{P}}$.*

Corollary 11 *[7] Let $\mathcal{P}$ be an acceptable general program. There exists a recurrent neural network $\mathcal{N}_r$ with semi-linear neurons such that, starting from an arbitrary initial input, $\mathcal{N}_r$ converges to a stable state and yields the unique fixpoint ($T_{\mathcal{P}} \uparrow \omega$ $(\mathbf{i})$) of $T_{\mathcal{P}}$, which is identical to the unique stable model of $\mathcal{P}$.*

Recall that, since $\mathcal{N}_r$ has semi-linear neurons, for each real value o_i in the output vector of $\mathcal{N}_r$, if $o_i \geq A_{min}$ then the corresponding i-th atom in $\mathcal{P}$ is assigned *true*, while $o_i \leq A_{max}$ means that it is assigned *false*.[6]

3 The Extended *C-IL²P* System

General logic programs provide negative information implicitly, using the closed-world assumption, while extended programs include explicit negation, allowing the presence of incomplete information in the database. "In the language of extended programs, we can distinguish between a query which fails in the sense that it does not succeed, and a query which fails in the stronger sense that its negation succeeds" [13]. The following example, due to John McCarthy, illustrates such a difference: a school bus may cross railway tracks under the condition that there is no approaching train. This can be expressed in a general logic program by the rule $cross \leftarrow \sim train$, only if the absence of $train$ in the database could be interpreted as the absence of an approaching train. Such a convention is unacceptable if the information about the presence of a train is not available. However, if we use classical negation and represent the above knowledge as the extended program: $cross \leftarrow \neg train$, then $cross$ will not be derived until the fact $\neg train$ is added to the database.

Therefore, the difference between $\neg p$ and $\sim p$ in a logic program is essential whenever we cannot assume that the available positive information about p is complete, that is, when the closed world assumption is not applicable to p. Nevertheless, the close-world assumption can be explicitly included in extended programs by adding rules of the form $\neg A \leftarrow \sim A$, whenever the information about A in the database is assumed to be complete. Moreover, for some predicates, the opposite assumption $A \leftarrow \sim \neg A$ may be appropriate.

[6] Activations in the interval $[A_{max}, A_{min}]$ are not allowed by the *Translation Algorithm*.

Definition 12 *(*Extended Logic Program*) An* extended logic program *is a finite set of clauses of the form $L_0 \leftarrow L_1, ..., L_m, \sim L_{m+1}, ..., \sim L_n$, where L_i $(0 \leq i \leq n)$ is a literal (an atom or the classical negation of an atom, denoted by $\neg$).*

The semantics of extended programs, called the *Answer Set Semantics*, is an extension of the stable model semantics for general logic programs. A "well-behaved" general program has exactly one stable model, and the answer that it returns for a ground query (A) is *yes* or *no*, depending on whether A belongs or not to the stable model of the program. "A 'well behaved' extended program has exactly one answer set, and this set is consistent. The answer that an extended program returns for a ground query (A) is *yes*, *no* or *unknown*, depending on whether its answer set contains A, $\neg A$ or neither. If a program does not contain classical negation, then its answer sets are exactly the same as its stable models" [13].

Consider, for example, the extended program $\mathcal{P} = \{\neg q \leftarrow \sim p\}$. Intuitively, it means: q is $false$ if there is no evidence that p is $true$. We will see that the only answer set for this program is $\{\neg q\}$ and, therefore, the answer that the program gives to the queries p and q are, respectively, $unknown$ and $false$.

Definition 13 *(*Definition *4* rewritten*) Let $\mathcal{P}$ be an extended program. By Lit we denote the set of ground literals in the language of $\mathcal{P}$. For any set $\mathcal{S} \subset Lit$, let $\mathcal{P}^+$ be the extended program obtained from $\mathcal{P}$ by deleting (i) each clause that has a formula $\sim L$ in its body when $L \in \mathcal{S}$, and (ii) all formulas of the form $\sim L$ present in the bodies of the remaining clauses.*

Following [4], we say that $\mathcal{P}^+ = \mathbf{R}_{\mathcal{S}}(\mathcal{P})$, which should read $\mathcal{P}^+$ is the *Gelfond-Lifschitz Reduction* of $\mathcal{P}$ w.r.t. $\mathcal{S}$ (after its inventors). By the above definition, $\mathcal{P}^+$ does not contain default negation ($\sim$), and its answer set can be defined as follows.

Definition 14 *(*Answer Sets of '$\sim$ free' Programs*) The* answer set *of $\mathcal{P}^+$ is the smallest subset $\mathcal{S}^+$ of Lit such that (i) for any rule $L_0 \leftarrow L_1, ..., L_m$ of $\mathcal{P}^+$, if $L_1, ..., L_m \in \mathcal{S}^+$ then $L_0 \in \mathcal{S}^+$, and (ii) if $\mathcal{S}^+$ contains a pair of complementary literals then $\mathcal{S}^+ = Lit$.*

Finally, the answer set of an extended program $\mathcal{P}$ that contains default negation ($\sim$) can be defined as follows.

Definition 15 *(*Answer Sets*) Let $\mathcal{P}$ be an extended program and $\mathcal{S} \subset Lit$. Let $\mathcal{P}^+ = \mathbf{R}_{\mathcal{S}}(\mathcal{P})$ and $\mathcal{S}^+$ be the answer set of $\mathcal{P}^+$. $\mathcal{S}$ is the* answer set *of $\mathcal{P}$ iff $\mathcal{S} = \mathcal{S}^+$.*

For example, the program $\mathcal{P} = \{\neg q \leftarrow \sim p\}$ has $\{\neg q\}$ as its only answer set, since no other subset of the literals in $\mathcal{P}$ has the same fixpoint property. As another example, compare the programs $\mathcal{P}_1 = \{\neg p \leftarrow\ ;\ p \leftarrow \neg q\}$ and $\mathcal{P}_2 = \{\neg p \leftarrow\ ;\ q \leftarrow \neg p\}$. Each of them has a single answer set: $\{\neg p\}$ and $\{\neg p, q\}$, respectively.[7]

Note that if $\mathcal{P}$ does not contain classical negation ($\neg$) then its answer sets do not contain negative literals. In other words, the answer sets of a general logic program are identical to its stable models. However, the absence of an atom A in the stable model of a general program means that A is $false$ (by default), while the absence of A (and $\neg A$) in the answer set of an extended program means that nothing is known about A.[8]

An extended logic program ($\mathcal{P}$) that has a consistent answer set can be reduced to a general logic program ($\mathcal{P}^*$) as follows. For any negative literal $\neg A$ occurring in $\mathcal{P}$, let A' be a positive literal that does not occur in $\mathcal{P}$. A' is called the *positive form* of $\neg A$. $\mathcal{P}^*$ is obtained from $\mathcal{P}$ by replacing all the negative literals of each rule of $\mathcal{P}$ by its positive form. $\mathcal{P}^*$ is called the positive form of $\mathcal{P}$. For example, the program $\mathcal{P} = \{a \leftarrow b, \neg c\ ;\ c \leftarrow\ \}$ can be reduced to its positive form $\mathcal{P}^* = \{a \leftarrow b, c'\ ;\ c \leftarrow\ \}$.

Definition 16 *For any set $\mathcal{S} \subset Lit$, let $\mathcal{S}^*$ denote the set of the positive forms of the elements of $\mathcal{S}$.*

Proposition 17 *[13] A consistent set $\mathcal{S} \subset Lit$ is an answer set of $\mathcal{P}$ if and only if $\mathcal{S}^*$ is a stable model of $\mathcal{P}^*$.*

The mapping from $\mathcal{P}$ to $\mathcal{P}^*$ reduces extended programs to general programs, although $\mathcal{P}^*$ alone does not indicate that A' represents the negation of A.

By Proposition 17, in order to translate an extended program ($\mathcal{P}$) into a neural network ($\mathcal{N}$), we can use the same approach as the one for general programs (Section 2.1), with the only difference that input and output neurons should be labelled as literals, instead of atoms. In the case of general logic programs, a concept A is represented by a neuron, and its weights indicate whether A is a positive or a negative literal in the sense of default negation, that is, the weights differentiate A from $\sim A$. In the case of extended logic programs, we must also be able to represent the concept $\neg A$ in the network. We do so by explicitly labelling input and output neurons as $\neg A$, while the weights differentiate $\neg A$ from $\sim \neg A$. Note that, in this case, both neurons A and $\neg A$ might be present in the same network.

Analogously to Theorem 9, the following proposition ensures that the translation of extended programs into neural networks is correct.

Theorem 18 *For each extended logic program $\mathcal{P}$, there exists a feedforward neural network $\mathcal{N}$ with exactly one hidden layer and semi-linear neurons such that $\mathcal{N}$ computes $T_{\mathcal{P}^*}$, where $\mathcal{P}^*$ is the positive form of $\mathcal{P}$.*

[7] Note that, as the answer set semantics assigns different meanings to the rules $p \leftarrow \neg q$ and $q \leftarrow \neg p$, it is not contrapositive w. r. t. $\leftarrow$ and $\neg$. The reason is that it interprets implications as inference rules or causal implication, rather than conditionals or material implication.

[8] Gelfond and Lifschitz think of answer sets as incomplete theories rather than three-valued models. Intuitively, answer sets are possible sets of beliefs that a rational agent may hold.

Proof. *By Definition 15, we simply need to rename each negative literal $\neg A_i$ $(0 \le i \le n)$ in $\mathcal{P}$ by A_i', and label the corresponding neuron in $\mathcal{N}$ as $\neg A_i$. Then, by Theorem 9, $\mathcal{N}$ computes $T_{\mathcal{P}^*}$.* $\square$

Example 19 *Consider the extended program $\mathcal{P} = \{A \leftarrow B, \neg C;\ \neg C \leftarrow B, \sim \neg E;\ B \leftarrow\ \sim D\}$.* C-IL^{2}P's *Translation Algorithm over the positive form $\mathcal{P}^*$ of $\mathcal{P}$ obtains the network $\mathcal{N}$ of Figure 2 such that $\mathcal{N}$ computes the fixpoint operator $T_{\mathcal{P}^*}$ of $\mathcal{P}^*$.*

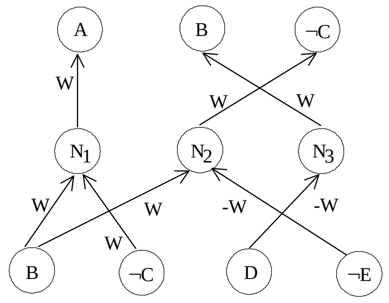

Fig. 2. From extended programs to neural networks.

As before, the network of Figure 2 can be transformed into a partially recurrent network by connecting neurons in the output layer (e.g., B) to its correspondent neuron in the input layer, with weight $W_r = 1$, so that $\mathcal{N}$ computes the upward powers of $T_{\mathcal{P}^*}$.

Definition 20 *(*Acceptable Extended Programs*) An extended logic program $\mathcal{P}$ is called an acceptable program if its positive form $\mathcal{P}^*$ is an acceptable program.*

Corollary 21 *Let $\mathcal{P}$ be a consistent acceptable extended program. Let $\mathcal{P}^*$ be the positive form of $\mathcal{P}$. There exists a recurrent neural network $\mathcal{N}$ with semi-linear neurons such that, starting from an arbitrary input, $\mathcal{N}$ converges to the unique fixpoint of $\mathcal{P}^*$ ($T_{\mathcal{P}^*} \uparrow \omega$), which is identical to the unique answer set of $\mathcal{P}$.*

Proof. *By Proposition 18, $\mathcal{N}$ computes $T_{\mathcal{P}^*}$. Assume that $\mathcal{P}$ is an acceptable program. By Corollary 11, when recurrently connected, $\mathcal{N}$ computes the unique stable model of $\mathcal{P}^*$. By Proposition 17, a consistent set $\mathcal{S} \subset Lit$ is an answer set of $\mathcal{P}$ if and only if $\mathcal{S}^*$ is a stable model of $\mathcal{P}^*$. Thus, $\mathcal{N}$ computes the unique answer set of $\mathcal{P}$.* $\square$

Example 22 *(Example 19 continued) Given any initial activation in the input layer of $\mathcal{N}$ (Figure 2), it always converges to the following stable state: $A = true, B = true, \neg C = true, D = false, \neg E = false$, which represents the answer set of $\mathcal{P}$, $\mathcal{S}_{\mathcal{P}} = \{A, B, \neg C\}$.*

Let us now briefly discuss the case of inconsistent extended programs. Consider, for example, the contradictory program $\mathcal{P} = \{B \leftarrow A;\ \neg B \leftarrow A;\ A \leftarrow\ \}$. As it is an acceptable program, its associated network always converges to the stable state that represents the set $\{A, B, \neg B\}$. At this point, we have to make a choice: either we adopt Gelfond and Lifschitz's Definition 14, and assume that the answer set of $\mathcal{P}$ is the set of all literals in the language (*Lit*), or we use a paraconsistent approach [2]. Since the presence of inconsistencies in *C-IL^{2}P* networks is tolerated (differently from classical logic, in which any inconsistency proves *Lit* and, thus, trivialises the theory), we believe that the second approach is more appropriate. "There is a need to develop a framework in which inconsistency can be viewed according to context, as a trigger for actions, for learning, and as a source of directions in argumentation."[10] This paper has paved the way for exciting new research on how to solve inconsistencies in incomplete theories using inductive learning in neural networks.

4 Conclusion and Future Work

This paper has presented an extension to the *C-IL^{2}P* system [5, 7] that defines a massively parallel model for extended logic programming, based on a single hidden layer neural network, capable of performing inductive learning from background knowledge and examples. This allows facts of commonsense knowledge to be represented more easily in the extended system, with the use of classical negation in addition to default negation.

The results presented in this paper indicate that neural networks, as (nonmonotonic) hybrid systems, and the research on Belief Revision [11] have some interesting interconnections. In principle, while neural networks present the advantage of eliciting new knowledge from examples, Belief Revision techniques can guarantee consistency of the new knowledge, and satisfy the (desirable) *Principle of Minimal Change*. A theoretical and empirical comparison of these two methods of theory refinement is long due, and would result in a highly attractive piece of work.

References

[1] R. Andrews, J. Diederich, and A. B. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-based Systems*, 8(6):373–389, 1995.

[2] H. Blair and V. S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[3] B. Boutsinas and M. N. Vrahatis. Artificial nonmonotonic neural networks. *Artificial Intelligence*, 132:1–38, 2001.

[4] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109:297–356, 1999.

[5] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125:155–207, 2001.

[6] A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. An analysis-revision cycle to evolve requirements specifications. In *Proc. 16th IEEE Automated Software Engineering Conference ASE01*, 2001.

[7] A. S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence Journal*, 11(1):59–77, 1999.

[8] A. S. d'Avila Garcez, G. Zaverucha, and V. N. Silva. Applying the connectionist inductive learning and logic programming system to power systems' diagnosis. In *Proc. IEEE ICNN97*, 121–126, Houston, 1997.

[9] M. Fitting. Metric methods: Three examples and a theorem. *Journal of Logic Programming*, 21:113–127, 1994.

[10] D. M. Gabbay and A. Hunter. Making inconsistency respectable: part 1. In *Fundamentals of AI Research*. Springer-Verlag, 1991.

[11] P. Gardenfors, editor. *Belief Revision*. Tracts in Theoretical Computer Science, Cambridge, 1992.

[12] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Logic Programming Symposium*, 1988.

[13] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[14] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[15] S. Holldobler and Y. Kalinke. Toward a new massively parallel computational model for logic programming. In *Proc. Workshop on Combining Symbolic and Connectionist Processing, ECAI 94*, 1994.

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

[17] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[18] W. Marek and M. Truszczynski. *Nonmonotonic Logic: Context Dependent Reasoning*. Springer-Verlag, 1993.

[19] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[20] M. O. Noordewier, G. G. Towell, and J. W. Shavlik. Training knowledge-based neural networks to recognize genes in DNA sequences. In *NIPS91*, vol. 3, 530–536, 1991.

[21] G. Pinkas. Reasoning, nonmonotonicity and learning in connectionist networks that capture propositional knowledge. *Artificial Intelligence*, 77:203–247, 1995.

[22] V. N. Silva, G. Zaverucha, and G. Souza. An integration of neural networks and nonmonotonic reasoning for power systems' diagnosis. In *Proc. IEEE ICNN95*, Perth, 1995.

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

[24] G. G. Towell and J. W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1):119–165, 1994.

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