First-order Logic Learning in Artificial Neural Networks

Mathieu Guillame-Bert, Krysia Broda and Artur d'Avila Garcez

***Abstract*— Artificial Neural Networks have previously been applied in neuro-symbolic learning to learn ground logic program rules. However, there are few results of learning relations using neuro-symbolic learning. This paper presents the system PAN, which can learn relations. The inputs to PAN are one or more atoms, representing the conditions of a logic rule, and the output is the conclusion of the rule. The symbolic inputs may include functional terms of arbitrary depth and arity, and the output may include terms constructed from the input functors. Symbolic inputs are encoded as an integer using an invertible encoding function, which is used in reverse to extract the output terms. The main advance of this system is a convention to allow construction of Artificial Neural Networks able to learn rules with the same power of expression as first order definite clauses. The system is tested on three examples and the results are discussed.**

I. Introduction

This paper is concerned with the application of Artificial Neural Networks (ANNs) to inductive reasoning. Induction is a reasoning process which builds new general knowledge, called hypotheses, from initial knowledge and observations, or examples. In comparison with other machine learning techniques, the ANN method is relatively good with noise, algorithmically cheap, easy to use and shows good results in many domains [2], [3], [18]. However, the method has two disadvantages: (i) efficiency depends on the initial chosen architecture of the network and the training parameters, and (ii) inferred hypotheses are not directly available – normally, the only operation is to use the trained network as an oracle.

To address these deficiencies various studies of ways to build the ANN and to extract the encoded hypotheses have been made [4], [11]. One method used in many techniques employs translation between a logic programming (LP) language and ANNs. Logic Programming languages are both expressive and relatively easily understood by humans. They are the focus of (observational predicate) Inductive Logic Programming (ILP) techniques, for example PROGOL [15], whereby Horn clause hypotheses can be learned which, in conjunction with background information, imply given observations. Translation techniques between ANNs and ground logic programs, commonly known as neural-symbolic learning [5], have been applied in [4], [9], [14]. Neural-symbolic computation has also been investigated in the context of non-classical and commonsense reasoning with promising results in terms of reasoning and learnability [6], [7]. Particular ANN architectures allow to simulate the "bottom-up" or forward derivation behaviour of a logic program. The reverse direction, though harder, allows to analyse an ANN in an understandable way. However, in order to perform induction in complex domains, an association between ANNs and more expressive first order logic is required. This paper presents such an association and is therefore a direct alternative to standard ILP techniques [15]. The expectations are to use the robustness and the highly parallel architecture to propose an efficient way to do induction on predicate logic programs.

Presented here is PAN (Predicate Association Network), which performs induction on predicate logic programs based on ANNs. More precisely, PAN learns first order deduction rules from a set of training examples consisting of a set of 'input atoms' and a set of 'expected output atoms'. This system allows to include initial background knowledge by way of a set of fixed rules which are used recursively by the system. Secondly, the user can give a set of variable initial rules that will be changed through the learning. In this paper we assume the set of variable initial rules is empty. In particular, we present an algorithm and results from some initial experiments of relational learning in two problem domains, the Michalski train domain [13] and a chess problem [16]. The algorithm employs an encoding function that maps symbolic terms such as f(g(a,f(b))) to a unique natural number. We prove that the encoding is injective and define its inverse, which is simple to implement. The encoding is used together with equality tests and the new concept of multi-dimensional neurons to facilitate term propagation, so that the learned rules are fully relational.

Section II defines some preliminary notions that are used in PAN. Section III presents the convention used by the system to represent and deal with logic terms. The PAN system is presented in Section IV and the results of its use on examples are shown in Section V. Due to space limitations only the most relevant parts of PAN are presented. For more details, please see [8]. The main features of the PAN system are summarised in Section VI and compared with related work. The paper concludes in Section VII with future work.

II. Preliminaries

This section presents two notions used in the system description in Section IV. They are *deduction rule*, introduced to define in a intuitive way rules of the form $Conditions \Rightarrow Conclusion$, and *Multi-dimensional neuron*, introduced as an extension of common (single dimensional) neurons.

A. Deduction rules

Definition 1: A *deduction rule*, or rule, consists of a conjunction of atoms and tests called the *body*, and an atom called

Mathieu Guillame-Bert, INRIA Rhône-Alpes Research Center, 655 Avenue de l'Europe, 38330 Montbonnot-St.-Martin, France; email: mathieu.guillame-bert@inrialpes.fr

Krysia Broda, Dept. of Computing, Imperial College, London SW7 2BZ; email: kb@imperial.ac.uk

Artur Garcez, Dept. of Computing, City Univeristy, London EC1V 0HB; email:aag@soi.city.ac.uk

the *head*, where any free variables (written $U, \ldots, Z$) in the head must be present in the body. When the body formula is evaluated as true, the rule is *applicable*, and the head atom is said to be *produced*. $\diamond$

Example 1: Here are some examples of deduction rules.

$$\begin{aligned} P(X,Y) \wedge Q(X) \Rightarrow & \quad R(Y) \\ P(g(X),Y) \Rightarrow & \quad P(X,Y) \\ P(X,Y) \wedge Q(f(X,Y),a) \Rightarrow & \quad R(g(Y)) \\ P(list(a,list(b,list(c,list(T,0))))) \Rightarrow & \quad S \\ P(X) \wedge Q(Y) \wedge X = Y \Rightarrow & \quad R \end{aligned}$$

In other words, a deduction rule expresses an association between a set of atoms and a singleton atom. To illustrate, if the rule $P(X,Y) \wedge Q(X) \Rightarrow R(Y)$ is applied to the set of ground atoms $\{P(a,b), P(a,c), Q(a), R(d)\}$ there are two expected atoms produced: $\{R(b), R(c)\}$.

Definition 2: A *test* $HU^n \rightarrow \{-1, 1\}$, where HU is known as the Herbrand universe and n is the test's arity, is said to succeed (fail) if the result is $1(-1)$. It is computed by a test module and is associated with a logic formula that is true for the given input iff the test succeeds for this input. $\diamond$

Equality ($=$) and inequality ($\neq$) are tests of arity 2. The function ($>$) is a test that only applies to integer terms.

In PAN, a learned rule is distributed in the network; for example, the rule $P(X,Y) \wedge P(Y,Z) \Rightarrow P(X,Z)$ is represented in the network as $P(X,Y) \wedge P(W,Z) \wedge (Y = W) \wedge (U = X) \wedge (V = Z) \Rightarrow P(U,V)$ and $P(X, f(X)) \Rightarrow Q(g(X))$ is represented as $P(X,Y) \wedge (Y = f(X)) \wedge (Z = g(X)) \Rightarrow Q(Z)$. This use of equality is central to the proposed system and its associated higher expressive power.

The allowed literals in the body and the head are restricted by the *language bias*, the set of syntactic restrictions on the rules the system is allowed to infer – eg to infer rules with variable terms only. The language bias gives a user some control on the syntax of rules that may be learned. This improves the speed of the learning and may also improve the generalization of the example – i.e. the language bias reduces the number of training examples needed.

Example 2: Suppose we assume the expected rule doesn't need functional terms in the body (it might be $P(X,Y) \Rightarrow R(X)$). The language bias used by PAN is 'no functional terms in the body' – call it b'. Next, assume the rule to be learned may have functional terms in the body (it might be $P(X, f(X)) \Rightarrow R(X)$). The new language bias b'' might be 'functional terms in the body with maximum deph of 1'. The user can specify a *relaxation function* for the transition from b' to b'' enabling PAN to relax the language bias from b' to b'' if it was unable to find a rule in one learning epoch.

B. Multi-dimensional neurons

PAN requires neurons to carry vector values i.e. multi dimensional values. Such neurons are called *Multi-dimensional neurons*. They are used to prevent the network converging into undesirable states. For instance, suppose three logical terms a, b and c are encoded as the respective integer values 1, 2 and 3 in an ANN, and for a given training example, suppose that c is an expected term. Let an ANN with regular single dimension neurons be presented with inputs 1, 2 and 3 and let it be trained to output 3. The ANN may produce the number 3 by adding 1+2, which is meaningless from the point of view of the indices of a, b and c. Use of the multi dimensional neurons avoids such unwanted combinations.

In what follows, let $\mathcal{V} = \{v_i\}_{i \in \mathbb{N}}$ be an infinite, normalized, free family of vectors, and $B_\mathcal{V}$ be the vector space generated by $\mathcal{V}$; i.e. $B_\mathcal{V} = \{x | \exists a_1, a_2, \cdots \in \mathbb{R} \wedge x = a_1 v_1 + a_2 v_2 + \ldots\}$.[1] Further, for all vectors $v \in B_\mathcal{V}$, some linear combination of elements of $\mathcal{V}$ equals v.

Definition 3: The *Set multi-dimension neuronal activation function multidim* : $\mathbb{R} \rightarrow B_\mathcal{V}$ is the neuron activation function defined by $multidim(x) = v_{\lfloor x \rfloor}$ $\diamond$

The inverse function is defined as follows.

Definition 4: The *Set single dimension activation function invmultidim*: $B_\mathcal{V} \rightarrow \mathbb{R}$ extracts the index of the greatest dimension of the input: $invmultidim(v) = i$ where $\forall j \; v_i \cdot v \geq v_j \cdot v$ $\diamond$

The property $\forall x \in \mathbb{R}, invmultidim(multidim(x)) = \lfloor x \rfloor$ follows easily from the definition and properties of $\mathcal{V}$. In the case of the earlier example, the term a (index 1) would be encoded as $multidim(1) = v_1$, b as v_2 and c as v_3. Since v_1, v_2 and v_3 are linearly independent, the only way to produce the output v_3 is to take the input v_3.

The functions $multidim$ and $invmultidim$ can be simulated using common neurons. However, for computational efficiency, it is interesting to have special neurons for those operations. The way to represent the multi-dimensional values, i.e. a vector value, is also important. In an ANN, for a given multi-dimensional neuron, the number of non-null dimensions is often very small. But the index of the dimension used can be important. For example, a common case is to have a multi-dimensional neuron with a single dimension non-null but with a very high index, for example $v_{10^{10}}$. It is therefore very important to not represent those multi dimensional values as an array indexed by the dimension number, but to use a mapping (like a simple table or a hash table) between the index of the dimension and its component.

Definition 5: The *Multi-dimensional sum activation function* computes the value of a multi dimentionnal neuron as follows: Let n be a neuron with the $multidimSum$ activation function, w_{nk} be the real value weight from k to n, and $value(n)$ be n's value (depending on the activation type of the neuron – eg multidim, multidimsum, sigmoid, etc.). Then

$$value(n) \quad = \quad \sum_{i \in I(n)} (w_{ni} \; value(i)) \; . \; \sum_{j \in S(n)} (w_{nj} \; value(j))$$

where $I(n)$ and $S(n)$ are respectively the sets of input of single dimensional and multi-dimensional neurons of n. If $I(n)$ or $S(n)$ is empty the result is 0. $\diamond$

III. Term encoding

In order to inject into, or extract terms from, the system, a convention to represent every logic term by a unique

[1] That is, $\forall i, j$, $i \neq j$, the properties $v_i \cdot v_i = 1$ and $v_i \cdot v_j = 0$ hold.

integer is defined. This section presents the convention used in PAN, in which $\mathcal{T}$ is the set of logical terms and $\mathcal{S}$ is the set of logical functors. The solution is based on Cantor diagonalization. It utilises the function $encode : \mathcal{T} \to \mathbb{N}$, which associates a unique natural number with every term, and the function $encode^{-1} : \mathbb{N} \to \mathcal{T}$, which returns the term associated with the input. This encoding is called the *Diagonal Term Encoding*. It is based on an indexing function $Index : \mathcal{S} \to \mathbb{N}$ which gives a unique index to every functor and constant. (The function $Index^{-1} : \mathbb{N} \to \mathcal{S}$ is the inverse of $Index$, i.e. $Index^{-1}(Index(\mathcal{S})) = \mathcal{S}$).

The diagonal term encoding has the advantage that composition and decomposition term operations can be defined using simple arithmetic operations on integers. For example, from the number n that represents the term $f(a, g(b, f(c)))$, it is possible to generate the natural numbers corresponding to the function f, the first argument a, the second argument $g(b, f(c))$ and its sub-arguments b, $f(c)$ and c using only simple arithmetic operations.

The encoding function uses the following auxiliary functions: $Index'$, which extends $Index$ to terms, E, which is a diagonalisation of $\mathbb{N}^2$ (figure 1), E', which extends E to lists, and E'', which recursively extends E'. The decoding function uses the fact that E is a bijection, see Theorem 1.

Example 3: Let the signature $\mathcal{S}$ consist of the constants a, b and c and functor f. Let $Index(a) = 1$, $Index(b) = 2$, $Index(c) = 3$ and $Index(f) = 4$.

The function $Index' : \mathcal{T} \to \mathbb{N}^+$ recursively gives a list of unique indices for any term t formed from $\mathcal{S}$ as follows.

$$Index'(t) = \begin{cases} [Index(t), 0] \text{ if } t \text{ is a constant} \\ [Index(f), Index'(x_0), ..., Index'(x_n), 0] \\ \quad \text{if } t = f(x_0, ..., x_n) \end{cases} \tag{1}$$

The pseudo inverse of $Index'$ is $(Index')^{-1}$:

$$(Index')^{-1}(L) = \begin{cases} f(t_1, \ldots, t_n) \\ \quad \text{where } L = [x_0, x_1, \ldots, x_n], \\ \quad f = Index^{-1}(x_0), \text{ and} \\ \quad t_i = (Index')^{-1}(x_i) \end{cases} \tag{2}$$

Example 4:

$$Index'(a) = [1, 0]$$
$$Index'(f(b)) = [4, [2, 0], 0]$$
$$Index'(f(a, f(b, c))) = [4, [1, 0], [4, [2, 0], [3, 0], 0], 0]$$

Now we define $E : \mathbb{N}^2 \to \mathbb{N}$, the *basic diagonal function*:

$$E(n_1, n_2) = \frac{(n_1 + n_2)(n_1 + n_2 + 1)}{2} + n_2 \tag{3}$$

Theorem 1: The basic diagonal function E is a bijection

Proof outline: The proof that E is an injection is straightforward and omitted here. We show E is a bijection by finding an inverse of E. Given $n \in \mathbb{N}$ we define as follows the inverse of E to be the total function $D : \mathbb{N} \to \mathbb{N}^2$

$$E^{-1}(n) = D(n) = (n1, n2) : \begin{cases} p &= \left\lfloor \frac{\sqrt{1+8.n}-1}{2} \right\rfloor \\ n_2 &= n - \frac{p.(p+1)}{2} \\ n_1 &= p - n_2 \end{cases} \tag{4}$$

where $p = n1 + n2$ is an intermediate variable and let $D_1(n) = n1$ and $D_2(n) = n2$.

It remains to show that D is indeed the inverse of E and p, $n1$ and $n2$ are all natural numbers. The proof uses the fact that for $n \in \mathbb{N}$ there is a value $k \in \mathbb{N}$ satisfying

$$\frac{k(k+1)}{2} \leq n < \frac{(k+1)(k+2)}{2} \tag{5}$$

(i) Show $E(n_1, n_2) = n$. Using $n_1 + n_2 = p$ and $n_2 = n - \frac{p(p+1)}{2}$ the definition of $E(n_1, n_2)$ gives $E(n_1, n_2) = \frac{p(p+1)}{2} + (n - \frac{p(p+1)}{2}) = n$

(ii) Show $p \in \mathbb{N}$. Let $k \in \mathbb{N}$ satisfy (5). Then $k \leq p < k + 1$ and hence $p = k$.

(iii) Show $n_2 \in \mathbb{N}$. By (4) $n_2 = n - \frac{p(p+1)}{2}$, and by (5) and Case (ii) $0 \leq n_2 < k + 1$.

(iv) Show $n_1 \in \mathbb{N}$. By (4) $n_1 = p - n_2$ and by (5) and Case (iii) $0 \leq n_1 \leq k$. •

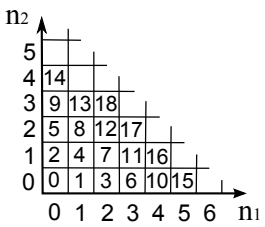


Fig. 1. $\mathbb{N}^2$ to $\mathbb{N}$ correspondence

Next, E' and D', the extensions of E and D in $\mathbb{N}^+$ are defined.

$$E'([n_0, ..., n_m]) = \begin{cases} n_0 \text{ if } m = 0 \\ E'([n_0, ..., n_{m-2}, E(n_{m-1}, n_m)]) \\ \quad \text{if } m > 0 \end{cases} \tag{6}$$

$$D'(n) = \begin{cases} [0] & \text{if } n = 0 \\ [D_1(n)] \,.D'(D_2(n)) & \text{if } n > 0 \end{cases} \tag{7}$$

Remark 1: In (7) "." is used for the list concatenation.

E'' and D'' are the recursive extensions of E' and D' in $\mathbb{N}^+$, defined by

$$E''(N) = \begin{cases} N, \text{ if } N \text{ is a number} \\ E'(E''(n_0), ..., E''(n_m)), \\ \quad \text{where } N = [n_0, ..., n_m], \text{ if } N \text{ is a list} \end{cases} \tag{8}$$

$$D''(n) = \begin{array}{l} [x_0, D''(x_1), ..., D''(x_{n-1}), x_n] \\ \quad \text{where } [x_0, ..., x_n] = D'(n) \end{array} \tag{9}$$

Remark 2: In (9), x_n is always equal to zero.

Next we define $encode : \mathcal{T} \to \mathbb{N}$ and $encode^{-1} : \mathbb{N} \to \mathcal{T}$:

$$encode(t) = E''(Index'(t)) \tag{10}$$
$$encode^{-1}(n) = (Index')^{-1}(D''(n)) \tag{11}$$

The uniqueness property of E can be extended to the full Diagonal Term Encoding that associates a unique natural number to every term; i.e. it is injective.

Example 5: Here is an example of encoding the term $f(a,b)$ with $index(a) = 1$, $index(b) = 2$ and $index(f) = 4$:

$$\begin{aligned} encode(f(a,b)) &= E''([4,[1,0],[2,0],0]) \\ &= E'([4,E''([1,0]),E''([2,0]),0]) \\ &= E'([4,1,3,0]) = E'([4,1,E(3,0)]) \\ &= E'([4,1,6]) = 775 \end{aligned}$$

To decode 775 requires first to compute $D'(775)$.

$$\begin{aligned} D'(775) &= [D_1(775)].D'(D_2(775)) \\ &= [4].D'(34) = [4,1,3,0] \end{aligned}$$

The final list $[4,1,3,0]$ is called the *Nat-List* representation of the original formula $f(a,b)$. Finally, $encode^{-1}(775) = (Index')^{-1}([4,D''(1),D''(3),0]) = f(a,b)$

The basic diagonal function E and its inverse E^{-1} can be computed with elementary arithmetic functions. This property of the Diagonal Term Encoding allows to extract the natural numbers representing the sub-terms of an encoded term. For example, from a number that represents the term $f(a,g(b))$, it is simple to extract the index of f and the numbers that represent a and $g(b)$.

Suppose E, D_1 and D_2 are neuron activation functions. Since D_1, D_2 and E are simple unconditional functions, they can be implemented by a set of simple neurons with basic activation functions – namely addition and multiplication operations and for D_1 and D_2 also the square root operation. It follows from (9), (11) and (13) that the encoding integer of the i^{th} component of the list that is encoded as an integer N is equal to $D_1(D_2^i(N))$. This computation is performed by an extraction neuron with activation function

$$extract^i(N) = \begin{cases} D_1(N) & \text{if } i = 0 \\ D_1(D_2^i(N)) & \text{if } i > 0 \end{cases} \quad (12)$$

For a given term F, if L is the natural number list representation of F and N its final encoding, then $extract^0(N)$ returns the functor of F and $extract^i(N)$ returns the i^{th} component of L. If the i^{th} component does not exist, $D_1(D_2^i(N)) = 0$. In the case that t is the integer that represents the term $f(x_1,...,x_n)$, i.e. $t = encode(f(x_1,...,x_n))$, then $extract^i(t) = encode(x_i)$ if $i > 0$ and $extract^0(t) = index(f)$ otherwise. It is possible to construct $extract^i$ using several simple neurons.

IV. Relational Network Learning : System PAN

The PAN system uses ANN learning techniques to infer deduction rule associations between sets of atoms as given in Definition 1. A training example $\mathcal{I} \Rightarrow \mathcal{O}$ is an association between a set of input atoms $\mathcal{I}$ and a set of expected output atoms $\mathcal{O}$. The ANN architecture used in PAN is precisely defined by a careful assembling of sub-ANNs described below. In what follows, the conventions used to load and read a set of atoms from PAN are given first. These are followed by an overview of the architecture of PAN and finally a presentation of the idea underlying its assembly is given.

The main steps of using PAN are the following:

Let $b \in \mathcal{B}$ be a language bias, and p be a relaxation function. Let $\mathcal{P}$ be a specific instance of PAN.

1) Construct $\mathcal{P}$ using bias b as outlined in Section IV-C.
2) Train $\mathcal{P}$ on the set of training examples i.e.
   - For every training example $I \Rightarrow O$ and every epoch i.e training step
     a) Load the input atoms I and run the network
     b) Compute the error of the output neurons based on the expected output atoms O
     c) Apply the back propagation algorithm
3) If the training converges return the final network $\mathcal{P}'$
4) Otherwise, relax b according to p
5) Repeat from step 2

A. Input convention

Definition 6: The *replication parameter* of a language bias is the maximum number of times any predicate can appear in the body of a rule. $\diamond$

The training is done on a finite set of training examples. Let Ψ_{in} (respectively Ψ_{out}) be the sets of predicates occurring in the input (respectively output) training examples and $\mathcal{P}$ be an instance of PAN. For every predicate $P \in \Psi_{in}$, the network associates r *activation input neurons* $\{a_{P_i}\}_{i\in[1,r]}$, where r is the *replication parameter* of the language bias. For every activation input neuron a_{P_i}, the network associates m *argument input neurons* $\{b_{P_{ij}}\}_{j\in[1,m]}$, where m is the arity of predicate P. Let $\mathcal{A} = \{a_{P_i}\}_{P\in\Psi_{in},i\in[1,r]}$ be the set of activation input neurons of $\mathcal{P}$. Let $I \Rightarrow O$ be a given training example and $I' = I \cup \{\Theta\}$, where Θ is an atom based on a reserved predicate that should never be presented to the system.

Let M be the set of all the possible injective mappings $m : A \to I'$, with the condition that m can map $a \in A$ to $e \in I'$ if the predicate of e is the same of the predicate bound to a, or if $e = \Theta$. The sequence of steps to load a set of input atoms I in $\mathcal{P}$ is the following:

1) Compute I' and M
2) For every $m \in M$
3) For all neurons $a_{P_i} \in \mathcal{A}$
   a) Set $value(a_{P_i}) = \begin{cases} 1 \text{ if } m(a_{P_i}) \neq \Theta \\ -1 \text{ if } m(a_{P_i}) = \Theta \end{cases}$
   b) For all j, set
   $$value(b_{P_{ij}}) = \begin{cases} encode(t_i) \text{ if } j \leq r \\ 0 \text{ if } j > r \end{cases}$$
   where $m(a_{P_i}) = P(t_1,\ldots,t_r)$

B. Output convention

For every predicate $P \in \Psi_{out}$, $\mathcal{P}$ associates an *activation output neuron*. For every activation output neuron, the network associates r *argument output neurons* $\{d_{P_i}\}_{i\in[1,r]}$, where r is the arity of the predicate P, $\mathcal{A}'$ is the set of output activation neurons of $\mathcal{P}$ and d_{P_i} is the i^{th} output argument neuron of P.

The following sequence of steps is used to read the set of atoms O from $\mathcal{P}$:

1) Set $O = \emptyset$
2) For all neuron $c_P \in \mathcal{A}'$ with $value(c_P) > 0.5$
   a) Add the atom $P(t_1, \ldots, t_p)$ to O, with $t_i = encode^{-1}(value(d_{P_i}))$

C. Predicate Association Network

This subsection presents the architecture of PAN, a non-recursive artificial neural network which uses special neurons called *modules*. Each of these modules has specific behavior that can be simulated by a recursive ANN. The modules are immune to backpropagation – i.e. in the case they are simulated by sub-ANNs, the backpropagation algorithm is not allowed to change their weights. [2]

The trainable part of PAN is non-recursive. The soundness and completeness of PAN learning depends on the training limit conditions, the initial invariant background knowledge and the language bias. For example, if we try to train it until all the examples are perfectly explained the learning might never stop. The worst case will be when PAN learns a separate rule for every training example.

There are five main module types; *Activation modules:* Memory and Extended Equality modules; *Term modules:* Extended Memory, Decomposition and Composition modules. In activation modules the carried values represent the activation level (as for a normal ANN), and in term modules the carried values represent logic terms through the term encoding convention. Each term module computes a precise operation on its inputs with a predicate logic equivalence. For example, the *decomposition module* decomposes terms – i.e. from the term $f(a, g(b))$, it computes the terms a, $g(b)$ and b. These five modules are briefly explained below. PAN is an assembly of these different modules into several layers, where each layer performs a specific task.

Globally, the organization of the modules in the PAN emulates three operations on a set of input atoms I.

1) Decomposition of terms contained in the atoms of I.
2) Evaluation and storage of patterns of tests appearing between different atoms of I. For example, the presence of an atom $P(X, X)$, the presence of two atoms $P(X)$ and $Q(f(X))$, or the presence of two atoms $P(X)$ and $Q(Y)$ with $X < Y$.
3) Generation of output atoms. The arguments of the output atoms are constructed from terms obtained by step 1. For example, if the term $g(a)$ is present in the system (step 1), PAN can build the output atom $P(h(g(a)))$ based on $g(a)$, the function h and the predicate P.

During the learning, the back-propagation algorithm builds the pattern of tests equivalent to a rule that explains the training examples.

Figure 2 represents informally the work done by an already trained PAN on a set of input atoms. PAN was trained on a set of examples (not displayed here) that forced it to learn the rule $P(X) \wedge P(f(X)) \Rightarrow Q(f(f(X)))$. The set of input atoms given to PAN is $\{P(a), Q(b), P(f(a))\}$.

[2]Backpropagation was chosen for its simplicity and because it is applied on a non recursive subset of PAN.

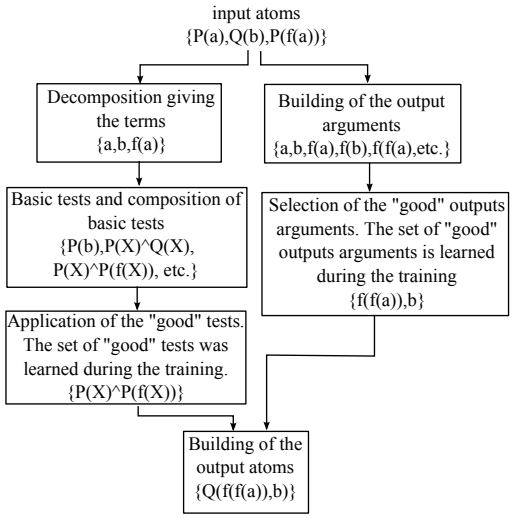


Fig. 2. Run of PAN after learning the rule $P(X) \wedge P(f(X)) \Rightarrow Q(f(f(X)))$

Below is a brief description of each module. The *Memory module* remembers if the input has been activated in the past during training. Its behavior is equivalent to a SR (set-reset) flip-flop with the input node corresponding to the S input.

The *Extended equality module* tests whether the natural numbers encoding two terms are equal. A graphical representation is shown in Figure 3. Notice that since 0 represents the absence of a term, the extended equality module does not fire if both inputs are 0.

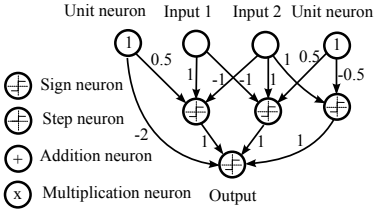


Fig. 3. Extended equality module

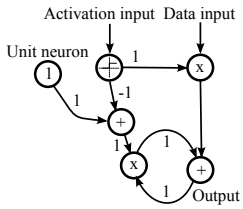


Fig. 4. Extended memory module

The *Extended memory module* remembers the last value presented on the *data input* when the *activation input* was activated. Its behavior is more or less equivalent to a processor register. Figure 4 shows a graphical representation.

The *Decomposition module* decomposes terms by implementing the $extract$ function of Equation (12).

The *Composition module* implements the $encode$ function of Equation (10).

D. Detailed example of run

This section presents in detail a very simple example running PAN including loading a set of atoms, informal interrelation of the neurons' values, recovery of the output atoms and learning though the back-propagation algorithm.

For this example, to keep the network small a strong language bias is used to build the PAN. The system is not allowed to compose terms or combine tests. It only has the

equality test over terms, is not allowed to learn rules with ground terms and is not allowed to replicate input predicates.

The initial PAN does not encode any rule. It has three input neurons a_{P_1}, $b_{P_{1,1}}$ and $b_{P_{1,2}}$, two output neurons c_R and d_{R_1} and input the set of atoms $\{P(a,b), P(b,f(b))\}$.

The injective mapping M has three elements $\{a_{P_1} \rightarrow P(a,b)$, $a_{P_1} \rightarrow P(b,f(b))$ and $a_{P_1} \rightarrow \Theta\}$. Loading of the input is made in three steps. Figure 5 gives the value of the inputs neurons over the three steps and Figure 6 shows a sub part of the PAN (only the interesting neurons for this example are displayed).

Step	1	2	3
a_{P_1}	1	1	0
$b_{P_{1,1}}$	$encode(a)$	$encode(b)$	0
$b_{P_{1,2}}$	$encode(b)$	$encode(f(b))$	0

Fig. 5. Input Convention mapping M

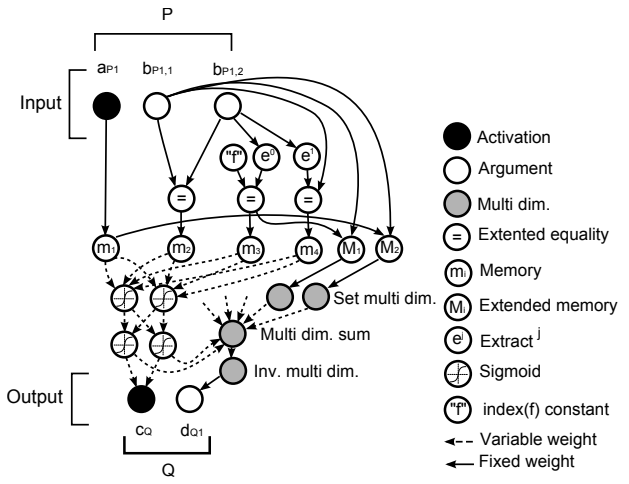


Fig. 6. Illustration of the PAN

Step	1	2	3
m_1	1	1	1
m_2	-1	-1	-1
m_3	-1	1	1
m_4	-1	1	1
M_1	0	encode(b)	encode(b)
M_2	encode(a)	encode(f(b))	encode(f(b))

Fig. 7. Memory and Extended memory module Values

Figure 7 gives the value of the memory and extended memory module after each step. The meaning of the different memory modules is the following one:
$m_1 = 1$: at least one occurrence of $P(X,Y)$ exists.
$m_2 = 1$: at least one occurrence of $P(X,Y)$ and $X = Y$.
$m_3 = 1$: at least one occurrence of $P(X,Y)$ and $Y = f(Z)$.
$m_4 = 1$: at least one occurrence of $P(X,Y)$ and $Y = f(X)$.
M_1 remembers the term Z when $P(X,Y)$ and $Y = f(Z)$.
M_2 remembers the term X when $P(X,Y)$ exists.

Suppose that the expected output atom is $Q(b)$. The expected output value of the PAN are $c_Q = 1$ and $d_{Q_1} = encode(b)$. The error is computed and the backpropagation algorithm is applied. The output argument d_{Q_1} will converge to be bound to M_1. After training on this example and depending on the other training examples, PAN could learn rules such as $P(X,Y) \Rightarrow Q(X)$ or $P(X,f(X)) \Rightarrow Q(X)$.

V. Initial Experiments

This section presents learning experiments using PAN. The implementation has been developed in C++ and run on a simple 2 GHz laptop. The first example shows the behavior of PAN on a simple induction problem. The second experiment evaluates the PAN on the chess endgame King-and-Rook against King determination of legal positions problem. The last run is the solution of Michalski's train problem. The final number of neurons and their architecture depends of the relaxation function chosen by the user. In all the experiments, a general relaxation function is used - i.e. all the constraints are relaxed one after another.

A. Simple learning

This example presents the result of learning the rule $P(X,f(X)) \wedge Q(g(X))) \Rightarrow R(h(f(X)))$ from a set of training examples. Some of the training and evaluation examples are presented in Figures 8 and 9. The actual sets contained 20 training examples and 31 evaluation examples. PAN achieved 100% success rate after 113 seconds running time with the following structure: 6 input neurons, 1778 hidden neurons and 2 output neurons; of the hidden neurons, 52% were composition neurons, 35% multi-dimension activation neurons, 6% memory neurons, 1% extended equality neurons.

$\{P(a,f(a)), Q(g(a))\} \Rightarrow \{R(h(f(a)))\}$
$\{P(b,f(b)), Q(g(b))\} \Rightarrow \{R(h(f(b)))\}$
$\{P(a,f(b)), Q(g(a))\} \Rightarrow \emptyset$
$\{P(a,b), Q(a)\} \Rightarrow \emptyset$
$\{P(a,a), Q(g(b))\} \Rightarrow \emptyset$
$\{P(a,a)\} \Rightarrow \emptyset$

Fig. 8. Part of the training set

$\{P(c,f(c)), Q(g(c))\} \Rightarrow \{R(h(f(c)))\}$
$\{P(d,f(d)), Q(g(d))\} \Rightarrow \{R(h(f(d)))\}$
$\{P(c,f(c))\} \Rightarrow \emptyset$
$\{P(d,f(d))\} \Rightarrow \emptyset$
$\{P(b,f(a))\} \Rightarrow \emptyset$
$\{P(a,f(b))\} \Rightarrow \emptyset$
$\{P(a,b)\} \Rightarrow \emptyset$

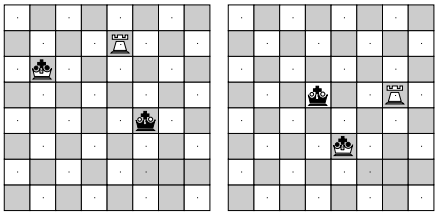
Fig. 9. Part of the evaluation set

B. Chess endgame King-and-Rook against King determination of legal positions problem

This problem classifies certain chess board configurations and was first proposed and tested on several learning techniques by Muggleton [16]. More precisely, without any chess knowledge rules, the system has to learn to classify 'King-and-Rook against King configurations' as legal (no piece can be taken) or illegal (one of the piece can be taken). Figure 10 shows a legal and an illegal configuration.

As an example, one of the many rules the system must learn is that the white Rook can capture the black King if their positions are aligned horizontally and the white King is not between them. Using Wkx as the x coordinate (column)

of the white King, Wry as the y coordinate (row) of the white Rook, etc., this rule can be logically expressed by $Game(Wkx, Wky, Wrx, Wry, Bkx, Bky) \wedge (Wry = Bky) \wedge ((Wky \neq Wry) \vee (Lt(Wkx, Wrx) \wedge Lt(Wkx, Bkx)) \vee (Lt(Wrx, Wkx) \wedge Lt(Bkx, Wkx))) \Rightarrow Illegal$.



A legal configuration An illegal configuration

Fig. 10. Chess configurations

For the evaluation of PAN a $20,000$ configurations database was used and three experiments made. For each experiment, a small part of the database was used to train the system and the remainder was used for the evaluation.

PAN is initially fed with the two fixed background knowledge predicates given in Muggleton's experiments [16], namely the predicate $X < Y \Rightarrow Lt(X, Y)$ and the predicate $(X + 1 = Y) \vee (X = Y) \vee (X = Y + 1) \Rightarrow Adj(X, Y)$.

Figure 11 shows the number of examples used to train the system, the number of examples to evaluate the system, the duration of the training and the success rate of the evaluations. In the three experiments, the produced PAN had 8 input neurons, 1 output neuron and 58 hidden neurons.

Experiment	1	2	3
Number of training examples	100	200	500
Number of evaluation test	19900	19800	19500
Training duration	4s	17s	43s
Success rate	91%	98%	100%

Fig. 11. King-and-Rook against King experiment's results

C. Michalski's train problem

The Michalski's train problem, presented in [13], is a binary classification problem on relational data. The data set is composed of ten trains with different features (number of cars, size of the cars, shape of the cars, objects in each car, etc.). Five of the trains are going East, and the five other are going West. The problem is to find the relation between the features of the trains and their destination. A graphical representation of the ten trains is shown in figure 12

To solve this problem, every train is described by a set of atoms. Figure 13 presents some of the atoms that describe the first train. The evaluation of PAN is done with "leave-one-out cross validation". Ten tests are done. For each of them, the system is trained on nine trains and evaluated on the remaining train. System PAN runs in an average of 36 seconds, and produces a network with 26 input neurons, 2591 hidden neurons and 1 output neuron; 50% of the hidden neurons are memory neurons and 47% are composition neurons. All the runs classify the remaining train correctly. The overall result is therefore a succes rate of 100%.

The system tried to learn a rule to build a predicate E, meaning that a train is going East. For a given train example, if the predicate is not produced, the train is considered as going West. This output predicate does not have any argument, hence the network used in this learning does not have any output argument-related layers. Moreover, none of the examples contain function terms and therefore none of the term composition and decomposition layers are needed. In this problem, the network only needed to learn to produce the atom E when certain conditions fire over the input atoms.

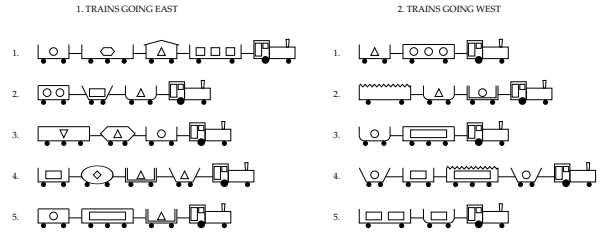


Fig. 12. Michalski's train problem

$Short(car2)$, $Closed(car2)$, $Long(car1)$, $Infront(car1, car2)$, $Open(car4)$, $Shape(car1, rectangle)$, $Load(car3, hexagon, 1)$, $Wheels(car4, 2)$,...

Fig. 13. Part of the Michalski's first train's description

VI. Related work

Inductive Logic Programming (ILP) is a learning technique for first order logic normally based on *Inverse Entailment* [15] and has been applied in many applications, notably in bioinformatics. Contrary to ILP, the behavior of PAN is more similar to an immediate consequence operator [9] but the system does not apply rules recursively, i.e. it applies only one step. As an example, let us suppose we present to the system a set of training examples that could be explained by the two rules $R_1 = \{P \Rightarrow Q, Q \Rightarrow R\}$. The PAN will not learn R_1, but the equivalent set of rules $R_2 = \{P \Rightarrow Q, P \Rightarrow R, Q \Rightarrow R\}$. It is expected that the neural networks in PAN will make the system more ammenable to noise than ILP and more efficient in general, but this remains to be verified in larger experiments.

Other ANN techniques operating on first order logic include those mentioned below.

Bader et al. present in [1] a technique of induction on T_P operators based on ANNs and inspired by [9] where propositional T_P operators with a three layer architecture are represented. An extension to predicate atoms is described with a technique called the Fine Blend System, based on the Cantor set space division. Differently from the Fine Blend, PAN uses the bijective mapping introduced earlier in this paper. We believe that this mapping is more appropriate for providing a constructive approach to first-order neuro-symbolic integration and more comprehensibility at the associated process of rule extraction.

Uwents and Blockeel describe in [17] a technique of induction on relational descriptions based on conditions on a set. This kind of induction is equivalent to induction on first order logic without function terms and without terms in the output predicates. The approach presented here is more expressive. Lerdlamnaochai and Kijsirikul present in [12] an

alternative method to achieve induction on logic programs without functions terms and without terms in the output predicates. The underlying idea of this method is to have a certain number of "free input variables", and present to them all the different combinations of terms from the input atoms. This is similar to propositionalisation in ILP and also less expressive than PAN.

Artificial Neural Networks are often considered as black-box systems defined only by the input convention, the output convention, and a simple architectural description. For example, a network can be defined by a number of layers and the number of nodes allowed in each of those layers. In contrast, the approach taken in PAN is to carefully define the internal structure of the network in order to be closely connected to first order structures. In addition, the following was made possible by PAN:

- the capacity to deal with functional (and typed) terms through the term encoding;
- the ability to specify the kinds of rule it is desired to learn through the language bias;
- the potential for extraction of learned rules, and
- the ability to generate output atoms with arbitrary term structures.

The last point is a key feature and seems to be novel in the area of ANNs. Similarly to the Inverse Entailment technique, PAN is able to *generate* terms, and not only to test them. For example, to test if the answer of a problem is $P(s)$, most relational techniques need to test the predicate P on every possible term (i.e. $P(a)$, $P(b)$, $P(c)$, $P(f(a))$, etc.) The approach followed in the PAN system directly generates the output argument. This guarantees that the output argument is unique and will be available in a finite time.

In contrast to ILP techniques and other common symbolic machine learning techniques ANNs are able to learn combinations of the possible rules. Consider the following example: if both the rules $P(X, X) \Rightarrow R$ and $Q(X, X) \Rightarrow R$ explain a given set of training examples, the PAN system is able to learn $P(X, X) \vee Q(Y, Y) \Rightarrow R$.

VII. Conclusion and Further Work

This paper presented an inductive learning technique for first order logic rules based on a neuro-symbolic approach [5], [7]. Inductive learning using ANNs has been previously applied mainly to propositional domains. Here, the extra expressiveness of first order logic can promote a range of other applications including relational learning. This is a main reason for the development of ANN techniques for inductive learning in first order logic. Also, ANNs are strong on noisy data, they are easy to parallelize, but they operate, by nature, in a propositional way. Therefore, the capacity to do first order logic induction with ANNs is a promising but complex challenge. A fuller description of the first prototype of PAN can be found in [8]. This section discusses some possible extensions and areas for exploration.

The typing of terms used in PAN is encoded architecturally in the ANN through the use of predicates. For example, a predicate P with arity two may accept as first argument a term of type *natural*, and as second argument a term of type *any*. However, it is currently not possible to type the sub-terms of a composed term. For example, whatever the type of $f(a, g(b))$, the types of a and $g(b)$ have to be *any*. This restriction on the typing may be inappropriate, and more general typing of networks may be important to study [10].

The first order semantics of rules encoded in the ANN is in direct correspondence with the network architecture. Future work will investigate extraction of the rules directly by the analysis of the weights of some nodes of the trained network. Knowledge extraction is an important part of neural-symbolic integration and a research area in its own right.

Further experiments will have to be carried out to evaluate the effectiveness of PAN on larger problems. and a comparison with ILP would also be relevant future work.

VIII. Acknowledgements

The authors would like to thank the anonymous referees for their constructive comments. The first author is supported by INRIA Rhône-Alpes Research Center, France.

References

[1] S. Bader, P. Hitzler, and S. Hölldobler, 'Connectionist model generation: A first-order approach', *Neurocomput.*, **71**(13-15), 2420–2432, (2008).
[2] H. Bhadeshia, 'Neural networks in materials science', *ISIJ International*, **39**(10), 966–979, (1999).
[3] C. M. Bishop, *Neural Networks for Pattern Recognition.*, Oxford University Press, Oxford, 1995.
[4] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach, 2001.
[5] A. S. d'Avila Garcez, K.Broda, and D. M. Gabbay, *Neural-Symbolic Learning Systems. Foundations and Applications*, Springer, 2002.
[6] A. S. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay, 'Connectionist modal logic: Representing modalities in neural networks', *Theoretical Computer Science*, **371**(1-2), 34–53, (2007).
[7] A. S. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay, *Neural-Symbolic Cognitive Reasoning*, Springer, 2008.
[8] M. Guillame-Bert, 'Connectionist artificial neural networks', *http://www3.imperial.ac.uk/computing/teaching/distinguished-projects*, (2009).
[9] S. S. Hölldobler and Y. Kalinke, 'Towards a new massively parallel computational model for logic programming', in *ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pp. 68–77, (1994).
[10] E. Komendantskaya, K. Broda, and A. d'Avila Garcez, 'Using inductive types for ensuring correctness of neuro-symbolic computations', in *Proc. of CiE2010, accepted for informal proceedings*, (2010).
[11] J. Lehmann, S. Bader, and P. Hitzler, 'Extracting reduced logic programs from artificial neural networks', in *Proc. IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy 05*, (2005).
[12] T. Lerdlamnaochai and B. Kijsirikul, 'First-order logical neural networks', *Int. J. Hybrid Intelligent Systems*, **2**(4), 253–267, (2005).
[13] R. S. Michalski, 'Pattern recognition as rule-guided inductive inference', in *Proc. of IEEE Trans. on Pattern Analysis and Machine Intelligence*, pp. 349–361, (1980).
[14] T. M. Mitchell, *Machine Learning*, McGraw-Hill, 1997.
[15] S.H. Muggleton, 'Inverse entailment and Progol', *New Generation Computing*, **13**, 245–286, (1995).
[16] S.H. Muggleton, M.E. Bain, J. Hayes-Michie, and D. Michie, 'An experimental comparison of human and machine learning formalisms', in *Proc. 6th Int. Workshop on Machine Learning*, pp. 113–118, (1989).
[17] W. Uwents and H. Blockeel, 'Experiments with relational neural networks', in *Proc. 14th Machine Learning Conference of Belgium and the Netherlands*, pp. 105–112, (2005).
[18] B. Widrow, D. E. Rumelhart, and M. A. Lehr, 'Neural networks: applications in industry, business and science', *Commun. ACM*, **37**(3), 93–105, (1994).