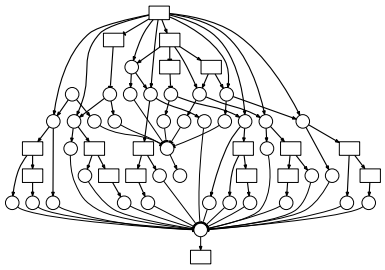
Imperial College London

Department of Computing

Connectionist Artificial Neural Networks



Master's Thesis

by

Mathieu Guillame-Bert

Supervised by Dr. Krysia Broda

Submitted in partial fulfillment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2009

Abstract

Because of the big complexity of the world, the ability to deal with uncertain and to infer "almost" true rules is an obligation for intelligent systems. Therefore, the research of solution to emulate *Inductive Reasoning* is one of the fundamental problem of Artificial Intelligence. Several approaches have been studied: the techniques inherited from the *Statistics* one side, or techniques based on *Logic* on the other side. Both of these families show complementary advantages and weakness. For example, statistics techniques, like decision trees or artificial neural networks, are robust against noisy data, and they are able to deal with a large quantity of information. However, they are generally unable to generate complexes rules. On the other side, Logic based techniques, like ILP, are able to express very complex rules, but they cannot deal with large amount of information.

This report presents the study and the development of an hybrid induction technique mixing the essence of statistical and logical learning techniques i.e. an *Induction* technique based on the *First Order Logic* semantic that generate hypotheses thanks to *Artificial Neural Networks* learning techniques. The expression power of the hypotheses is the one of the predicate logic, and the learning process is insensitive to noisy data thanks to the artificial neural network based learning process. During the project presented by this report, four new techniques have been studied and implemented: The first learns propositional relationship with an artificial neural network i.e. induction on propositional logic programs. The three other learn first order predicate relationships with artificial neural networks i.e. induction on predicate logic programs. The last of these techniques is the more complete one, and it is based on the knowledge acquired during the development of all the other techniques.

The main advance of this technique is the definition of a convention to allow the interaction of predicate logic programs and artificial neural networks, and the construction of Artificial Neural Networks able to learn rule with the predicate logic power of expression.

Contents

Chapter 1

Introduction

Definition 1.1. *Deduction*, or *deductive reasoning*, is a reasoning process which build new knowledge, called conclusion, from an initial knowledge. For example, a person that believe that cakes are alway good, and that see a cake in a bakery store, will conclude that this precise cake is good. The process of deduction does not make suppositions, therefore, if the initial knowledge is true, the conclusion is alway true.

Definition 1.2. *Induction*, or *inductive reasoning*, is a reasoning process which build new general knowledge, called hypotheses, from an initial knowledge and some observation. For example, a person that always eat good cake, may believe that cakes are always good. The process of induction does make suppositions, therefore, even if observations and the initial knowledge is correct, the hypothesis may be wrong.

Definition 1.3. *Abduction*, or *abductive reasoning*, is a reasoning process which build new particular knowledge, called supposition, from an initial knowledge and some observation. For example, a person that believe that all cake are alway good and that is tasting a good food without seeing it, may consider that this food is a cake. Like the induction, the process of abduction does make suppositions, therefore, even if observations and the initial knowledge is correct, the supposition way be wrong.

Induction sometimes refers to induction and abduction according to the previous definition.

Machine learning is a sub-part of the Artificial Intelligence field concerned with the development of knowledge generalization methods i.e. Inductive methods. A typical machine learning method infers hypotheses on a domain from examples of situations.

Today, since the development of computers, ways of algorithms have been developed, and successfully used in a large range of domains (image analysis, speech recognition, medical analysis, etc.) . However, there is currently not an ultimate technique, but a lot of different techniques with, for each of them, advantages and inconveniences (noise sensibility, expression power, algorithmic complexity, soundness, completeness, etc). The research in this area is very active.

1.1 Symbolic and connectionist knowledge connection

The Artificial Neural Networks (ANN) method is one often used technique. In comparison with other machine learning technique, ANN method is relatively insensible to noise, it is algorithmically cheap, it is easy to use and it shows good results in a lot of domains . However, this method has two major disadvantages: First, the efficiency depends on the initial chosen architecture of the network and the training parameters (which are not always obvious). Secondly, the inferred hypotheses are not directly available i.e. the only operation is to use the trained network as an oracle.

In order to address these deficiencies, various studies of different ways to build the artificial neural network and to extract the encoded hypotheses have been made [**?**, **?**]. One of the methods used in many of those techniques uses translation between a logic programming (LP) language and ANNs.

Logic Programming languages are powerful ways of expression, and are relatively easily understandable by humans. Based on this observation, the use of translation techniques between LP languages and ANNs seems to be a good idea. The translation from logic programs to ANNs is an easy way of specification of neural networks architecture (Logic languages → neural network). And the translation from ANNs to logic programs gives an easy way to analyze trained neural network (neural network → Logic languages).

Since ANN are naturally close to propositional implication rules, the translations between ANN and propositional logic programs is easier than the translation between ANN and predicate logic programs.

The association of the translations from ANN to logic programs and from logic programs to ANN allows performing induction on logic programs. It is therefore a direct alternative solution to standard ILP techniques [**?**]. The expectations are to use the robustness, the high parallel architecture and the statistically

distributed analyse of artificial neural networks techniques, to propose an efficient way to do induction on logic programs.

Translation methods between ANN and propositional logic programs have been successfully studied and developed for a long time [**?**, **?**, **?**, **?**]. The second chapter of this paper is precisely a non exhaustive presentation and analysis of some of those techniques. Several attempts to extend those translation techniques to predicate logic programs have been made. However, because of the higher complexity, there is still a lot of work to do [**?**]. The third chapter of this thesis introduce this problem and presents several important notions needed for the following chapters. Three attempts to solve this problem based on completely different approaches have been explored during this project. Each attempt is presented in a different chapter as a technique. The main goal of the two first technique is to introduce the basic notions of the third technique. For each of the techniques, a presentation is done, running examples are given, and a discussion that introduce the next technique is presented. The final chapter discusses this work and presents some conclusions.

More precisely, the achievements of this work are the following ones. The study and the implementation of several techniques of *Inductive reasoning* on *Propositional logic* based on *Artificial Neural Networks*. The study and the implementation of several techniques of Propositional rule extraction from Artificial Neural Networks. The development of several mathematical tools used in this research. And finally, the development, the implementation and the testing of three techniques of Inductive reasoning on First Order Logic based on Artificial Neural Networks.
At the sequel is a reminder of the different notions used in this document.

1.2 Artificial Neural Networks

Artificial Neural Networks (or ANN) are mathematical objects inspired from neuroscience research that try to imitate some key behaviours of animal brains (biological neural networks) . Several models of ANN exist. They are often based on a more a less accurate statistical model of natural neurons. However, some ANNs are intentionally not inspired from animal brains .

The main advantages of ANN are their capacity to "learn" a solution to a general problem by only analysing some instance of resolution (training examples), and to be insensitive to noisy data i.e. even if some input or training data are corrupted, ANNs produce good results.

In this paper we will use one of the simplest versions of ANN: the attractor artificial neural networks. This model is actually extremely simple. It is often used to do data classification or function approximation.

1.2.1 Definition

An artificial neural network can be represented by a directed graph with nodes N and edges E such that for every node $n \in N$ there is an "activation function", and for every edge $e \in E$ there is a "weight".

A node without input connections is called an "input node" and a node without output connections is called a "output node". Nodes with input and output connections are called "hidden nodes".

Every hidden and output node has a value that depends on the value on the other nodes connected to them.

In the case of input nodes, the values are directly defined by the user (input of the network).

The value of every hidden and output node is defined by the following equation:

$$V(n) = f_n(\sum_{i \in InputConnectionNodes(n)} w_{i \to n}.V(i))$$

Remark 1.1. $w_{i \to n}$ refers to the weight between the node i and the node n which is sometime written as $w_{n,i}$.

With $V(n)$ the value of the node n, $InputConnectionNodes(n)$ the set of all node connected to n, $w_{i \to n}$ the "weight" of the connection from i to n and f_n the activation function of n.

The value $V(n)$ of a neuron usually belongs to $\mathbb{R}$, the set of real numbers, but it can be any other kind of information (integer, multi-dimensional spaces, etc.). However, the basic learning methods need the activation function to be continuous and therefore the information space to be compact.

The figure 1.1 is a graphical representation of the following ANN. It has two input neurons, one output neuron and one hidden layer with three hidden neurons. The activation function and the different weights are not represented.

$$N = \{I_1, I_2, H_1, H_2, H_2, O_1\}$$
$$E = \{(I_1, H_1), (I_1, H_2), (I_1, H_3), (I_2, H_1), (I_2, H_2), (I_2, H_3), (H_1, O_1), (H_2, O_1), (H_3, O_1)\}$$

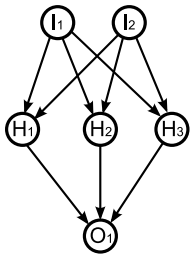


Figure 1.1: an example of small ANN

The most commonly used activation functions are the sigmoid and bipolar sigmoid functions, the hyperbolic tangent function, the sign function, and the step function. The figure 1.2 represents some of those functions.

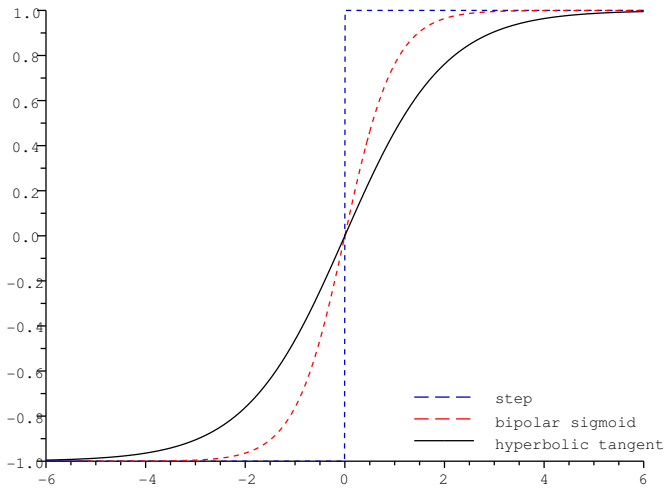


Figure 1.2: Common ANN activation functions

1.2.2 Construction of an ANN

Definition 1.4. A trained ANN for a problem P gives for a given input instance of P presented to the input neurons, an approximation of the associated solution presented from output neurons.

The usual way to use a trained ANN is to define values at the input nodes, and to read at the values produced at the output nodes.

To solve a given problem, the usual procedure is to define a blank ANN with a chosen architecture and to set all the initial weight to random values close to 0; then to train it on instances of input and expected outputs; and finally to use it to compute outputs on new input data.

The definition of the architecture and the training process are two complex operations. Several papers like give a deeper introspection in this area.

However, a common way to specify an ANN is to define a number of hidden layers and a number of neuron for each of those layers. All the neurons of a given layers i are connected to all the neurons of the next layer $i+1$. If i is the last layer, all the neurons of the layer i are connected to all the output neurons. And all the neurons of the input layer are connected to all the neuron of the first hidden layer.

1.2.3 Extended back propagation algorithm

In this paper only non recurrent ANNs will be used. The used learning algorithm is called "the extended back propagation algorithm". It is an extension of the back propagation algorithm with extra information for edges. The used values are real ($\mathbb{R}$) and multi-dimensional real spaces ($\mathbb{R}^*$) . The used activations functions are addition, multiplication, bipolar sigmoid and square root.

For this algorithm we add two new characteristics to every edge. An edge can either be a *variable* edge, a *protected* edge, a *stop back propagation* edge, or a *protected* and *stop back propagation* edge. Additionally, a weight of a edge can be limited with a lower and/or an upper limit. Variable edge behaviour is the same as

edge in the common back propagation algorithm. The weight of a protected edge cannot change. The error spread (first step of the back propagation algorithm) does not go "through" the "stop back propagation" edges.

The back propagation algorithm is defined as follow:

The training examples $\{T_i\}_{i\in\{1,maxExample\}}$ are pairs of $<$ input values, expected output values $>$. The parameters of the back propagation are the learning rate (ϵ) and the number of epochs ($maxEpoch$) i.e. the number of training rounds. The number of epoch condition can be replaced by various other conditions based on the learning.

1. For $i \in \{1, maxEpoch\}$
   (a) For $j \in \{1, maxExample\}$
      i. Load the input values $\{I_{j,k}\}_{k\in\mathbb{N}}$ for the training example $T_j = (\{I_{j,k}\}_{k\in\mathbb{N}}, \{O_{j,k}\}_{k\in\mathbb{N}},)$ in the network
      ii. For all output neurons n_l
         A. Calculate the "error" of n_l as the difference between the expected value of n_l and the value of n_l
         $error(n_l) = O_{j,l}$
      iii. For all input and hidden neurons n_l
         A. Calculate the "error" of n_l as with:
         $error(n_l) = \sum_{m\in OutputConnectionNodes(n_l)} error(n_m).w_{n_l\to n_m}.\delta_{n_l\to n_m}$
      iv. For all edge $e_{l\to m}$ between the neuron n_l and n_m that is not "protected"
         A. Increase the weight of the edge by Δw with:
         $\Delta w = \epsilon.V(n_l).error(n_m).\frac{df_m(x)}{d(x)}(S(n_l))$
         B. If the weight w is lower than lower its limite l
         Set $w = l + s$, with $s \in [0, \epsilon]$ a small random value
         C. If the weight w is greater than its upper limite L
         Set $w = L - s$, with $s \in [0, \epsilon]$ a small random value

Where $OutputConnectionNodes(n)$ the set of all node connected from n, $\delta_{n_l\to n_m} = 0$ if the edge between n_l and n_m is a "stop back propagation edge", and $\delta_{n_l\to n_m} = 1$ otherwise , and f_n the activation function of n.

The figure 1.3 is a graphical representation of the the small artificial neural network presented figure 1.1, extended with this new characteristic.

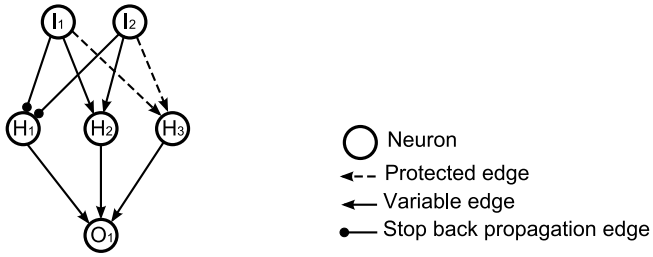


Figure 1.3: an example of small ANN

1.3 Logic Programs

Not all the notions of logic programming are presented here. We develop terms and notations needed to ensure a correct understanding of the following parts [**?**].

1.3.1 Propositional Logic programs

Definition 1.5. A *literal* is a propositional term or the negation of a propositional term.

Definition 1.6. A *disjunction* is a list of literals separated by the *Or* ($\vee$) operator.

Definition 1.7. A clause is a disjunction of literals.

Example 1.1.

$$A \vee B \vee \neg C \tag{1.1}$$

Definition 1.8. A *Horn clause* is a clause with at most one positive literal. To help the reading, they are written in the following equivalent form:

Clause	Used notation
$\neg a_1 \vee ... \vee \neg a_n$	$\bot \leftarrow a_1 \wedge ... \wedge a_n$
$b \vee \neg a_1 \vee ... \vee \neg a_n$	$b \leftarrow a_1 \wedge ... \wedge a_n$
b	$b \leftarrow$ (or $b \leftarrow \top$)

The b (when it is present) is called the head of the clause.
The $a_1, ..., a_n$ are called the body of the clause.

Definition 1.9. A *definite Horn clause* is a Horn clause with exactly one positive literal.

Definition 1.10. A *propositional logic program* is a set of clauses.

Definition 1.11. A *consequence operator* is a mapping operator from a set of formulas to a set of formulas.

Definition 1.12. The Herbrand base of a propositional logic program P is the set of all propositional terms of P.

Definition 1.13. Let P be a logic program, B_P the Herbrand base of P, and I be a Herbrand interpretation of P. The T_P operator (Immediate consequence operator) is a consequence operator and is a mapping of Herbrand interpretations defined in the following way:

$$T_P(I) = \{\alpha \in B_P | \alpha \leftarrow a_1, ..., a_n, \neg b_1, ..., \neg b_n \in P \text{ with } \forall a_i, a_i \in I \text{ and } b_i \text{ with } \forall b_i, b_i \notin I\}$$

1.3.2 Predicate Logic programs

Predicate logic programs allow the use of discrete variables and function in the syntax. The notions of Horn clauses, definite Horn clauses and consequence operators can be directly extended.

Remark 1.2. In a predicate logic clause all variables have to be quantified ($\exists$ or $\forall$). When the quantifier is not written, the quantifier $\forall$ is supposed.

Definition 1.14. A *constant* is a function of arity 0.

Definition 1.15. A *ground clause* is a clause that does not contains any variable.

Definition 1.16. The Herbrand universe of a predicate logic program P is the set of all ground terms of P.

Definition 1.17. The Herbrand base of a predicate logic program P is the set of all ground atoms of P.

Definition 1.18. Let P be a logic program, B_P the Herbrand base of P, and I be a Herbrand interpretations of P. The T_P operator (immediate consequence operator) is a consequence operator is a mapping of Herbrand interpretations defined in the following way:

$$T_P(I) = \{\alpha \in B_P | \alpha \leftarrow a_1, ..., a_n, \neg b_1, ..., \neg b_n \text{ is a ground instance of a clause of } P \text{ , with} \forall a_i, a_i \in I \text{ and } \forall b_i, b_i \notin I\}$$

Remark 1.3. Every propositional logic program is also a predicate logic program.

1.3.3 Higher level Logic programs

Higher level logics exist. A logic L is higher that the predicate logic if all notions of predicate logic can be expressed in L, and some notions of L cannot be expressed with the predicate logic.

Examples of higher logic can be logics that accepts predicates of predicates, functions of functions or allowing real number as terms.

A higher level logic that allows equality over function is used in a sound extraction of logic programs from ANNs. Those extracted logic programs can be reduced to predicate logic programs. However, this transformation causes the lost of the soundness of the extraction.

Here is two examples of formula of this logic.

$$P([F](X,Y)) \wedge Q(X) \wedge (([F] = f) \vee ([F] = g)) \rightarrow R(Y)$$
$$P([F](X,Y)) \wedge Q(X) \wedge ([F] \neq f) \rightarrow R(Y)$$

Where $[F]$ is a meta function.

The first formula can be equivalently rewritten into two predicate logic programs. The second formula cannot be equivalently rewritten into predicate logic programs.

1.4 ILP

Inductive logic programming (ILP) is a Machine Learning domain that uses Logic Program (LP) to represent example, background knowledge and hypothesis. In contrast to the techniques presented in this document, the ILP does not use connectionist intermediate representation but directly deal with logic programs.

Formally, given a set of examples E and a initial background knowledge B (both presented as a set of clauses), the ILP is looking for ways to find the hypothesis H such that $B \wedge H \models E$ while maximizing the probability $P(H|E)$ (the more probable hypothesis given the example, that explains the examples given the background knowledge).

Several approaches exist to generate hypotheses (inverse resolution , relative least general generalizations , inverse implication , and inverse entailment, etc.).

1.4.1 Inverse Entailment

Inverse Entailment (IE) is one of the approach used to generate hypothesis. It was initially explored by Mr. Muggleton through the *Progol* implementation [**?**]. It is based on the following consideration.

Suppose E, B and H representing a set of example, the background knowledge and the hypothesis such that $B \wedge H \models E$. Therefore $B \wedge \neg E \models \neg H$ holds.

Let's define $\bot(E,B) = \{\neg L | L \text{ is a literal and } B \wedge \neg E \models L\}$. $B \wedge \neg E \models \neg\bot(E,B) \models \neg H$ holds.

The hypothesis is built by generalizing $\bot(E,B)$ according to the formula $H \models \bot(E,B)$.

Chapter 2

Induction on Propositional Logic Programs through Artificial Neural Network

In this chapter is presented two of the techniques used to "translate" some fragments of Propositional Logic programs into Artificial Neural Networks.

2.1 Training ANN on propositional logic programs

2.1.1 The KBANN Algorithm

The KBANN (Knowledge Based Artificial Neural Network) algorithm is the first developed learning algorithm able to use initial domain theory to generate hypothesis (machine learning problem). It was described by Shavlik and Towell in 1989 [**?**].

Informally, the algorithm takes as input a definition of a domain theory and a set of training example. The generated output is a revised version of the domain theory that fits the examples. The interesting idea is that, thanks to the initial given domain theory, the search for a hypothesis is improved i.e. if some examples are not well explained by the domain theory, this one will be changed until all the examples are taken in account. If the initial domain theory perfectly fits the training examples, the returned domain theory will be the initial domain theory.

Definition 2.1. A learning problem is given by the tuple $< P, E >$, where P is a non-recursive definite Horn program, and E a set of $<$ input , expected output $>$ examples.

Formally, the input domain theory is a set of propositional, no recursive, definite Horn clauses, the examples are $<$ input , expected output $>$ examples, and the returned output domain theory is a Artificial Neural Network.

2.1.1.1 Algorithm

Here is a pseudo code version of the KBANN algorithm.

Algorithm 1 KBANN(Initial back ground knowledge, Training Examples)

1. Rewrite rules so that disjunctions are expressed as a set of rules that each have only one antecedent.
2. For each atom A that are present in more than one head of a clause of the Domain-Theory, create a network neuron u as follows:
3. Set the threshold weight for the neuron u to $\frac{W}{2}$.
   (a) For each clause c with A in the head, do ($c = A \leftarrow B_i$):
      i. Connect the inputs of the neuron u to the attributes B_i.
      ii. If B_i is a positive literal, assign a weight of W to the edge.
      iii. If B_i is a negative literal, assign a weight of $-W$ to the edge.
4. For each Horn clause in the Domain-Theory that are not used in the previous step, create a network neuron u as follows:
   (a) Connect the inputs of this neuron to the attributes tested by the clause antecedents.
   (b) For each non-negated antecedent of the clause, assign a weight of W to the corresponding sigmoid unit input.
   (c) For each negated antecedent of the clause, assign a weight of $-W$ to the corresponding sigmoid unit input.
   (d) Set the threshold weight for this neuron to $-(n - 0.5)W$, where n is the number of non-negated antecedents of the clause.
5. Compute the depth of every neuron.
6. Add additional connections among the network units, connecting each network unit at depth i to all network neurons at depth $i + 1$. Assign random near-zero weights to these additional connections.
7. Apply the BACKPROPAGATION algorithm to adjust the initial network weights to fit the Training-Examples.
8. The BACKPROPAGATION is described in the paper

Definition 2.2. The depth of a neuron is the size of the longest path between this neuron and an input neuron.

The only permitted input terms are those that are not defined by a head of a clause (for each of them there is a corresponding input node). The *true* and $false$ values are respectively expressed by a value of 1 and 0 in the neural network. The *unknown* value is not allowed.

The basis of the algorithm is to translate the Horns clauses into an Artificial Neural Network and to train it on the given training examples with the BACKPROPAGATION [**?**] technique.

The W parameter give an initial scaling of numbers used in the artificial neural network. Its value should be fixed depending of the way the machine that runs the algorithm deal with numbers. The value 1 is generally a good choice.

2.1.1.2 An example

Suppose the following logic program is given as a domain theory:

$$P = \begin{cases} bird & \leftarrow ostrich \\ bird & \leftarrow sparrow \\ bird & \leftarrow woodpecker \\ bird & \leftarrow dove \\ bird & \leftarrow penguin \\ fly & \leftarrow bird \wedge \neg exception \\ fly & \leftarrow balloon \\ exception & \leftarrow ostrich \end{cases}$$

If considered as a domain theory for the real world, it is wrong; as unlike this program's implication, penguins cannot really fly.

During the first step (1), the algorithm rewrites the program P in the following way:

$$P' = \begin{cases} bird & \leftarrow ostrich \\ bird & \leftarrow sparrow \\ bird & \leftarrow woodpecker \\ bird & \leftarrow dove \\ bird & \leftarrow penguin \\ fly' & \leftarrow bird \wedge \neg exception \\ fly & \leftarrow fly' \\ fly & \leftarrow balloon \\ exception & \leftarrow ostrich \end{cases}$$

And the generated artificial neural network is:

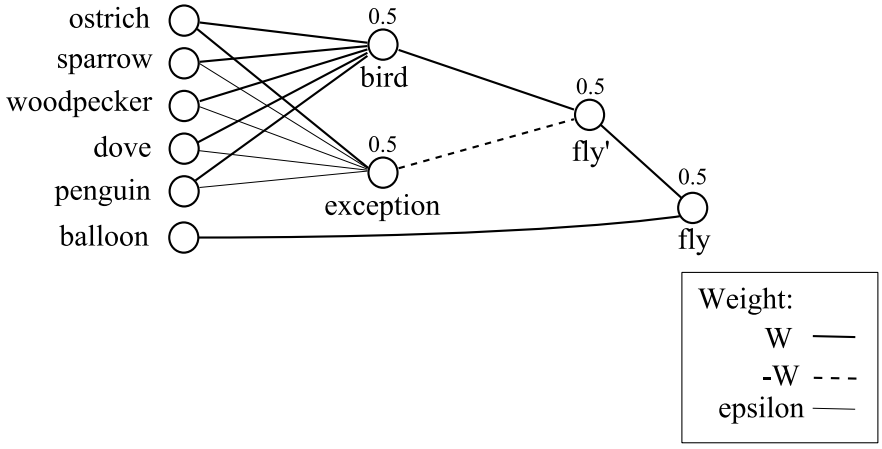


Figure 2.1: Initial domain theory

Remark 2.1. The value displayed on top of the neurons are the threshold value i.e. the negation of the weight of a connection from the unit neuron, to this neuron. The thresholds of *bird*, *exception* and fly' are equal to $-0.5 \cdot W$.

Assume that the training example set contains at least an example of *penguin* which are actually not able to fly and no sample with a flying penguin. Then after the training (BACKPROPAGATION step), the final domain theory is corrected and the neural network will become:

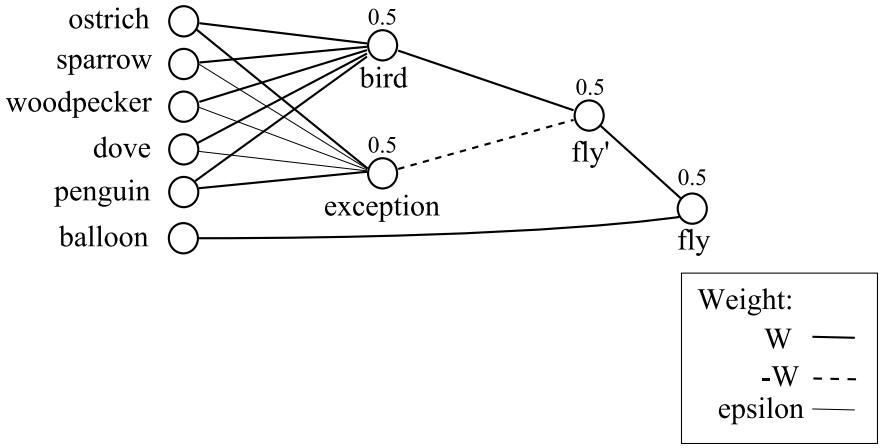


Figure 2.2: Final domain theory

2.1.1.3 Applications

In 1990 , Towell used the KBANN algorithm on DNA segment implication. Based on a (incomplete) domain theory, the KBANN algorithm output neural network had an error rate of $\frac{4}{106}$, when a direct artificial neural network trained with the BACKPROPAGATION algorithm displayed an error rate of $\frac{8}{106}$.

This experiment is related in the book "Machine Learning" [**?**]. However, it's important to note that the definition of the structure of the artificial neural network can have important consequences on its performances. Therefore to give a signification to this result, it is important to analyze the actual architecture of the neural network that competed with the KBANN algorithm. In fact, all the interest of the KBANN algorithm is in defining the structure of the artificial neural network.

2.1.1.4 Remarks

Even if the KBANN algorithm is based on a very interesting idea, it suffers from several important lacks as a machine learning algorithm.

First of all, the limitation of the initial domain theory to the propositional, non recursive definite Horn clause is a strong limitation to the kind of domain the KBANN algorithm can be applied on. Most of human interesting problems need at least the expression of free variables.
Nevertheless, it is important to note that in the case of finite Herbrand model, first order, non recursive Horn clauses can be handled by instantiating the clauses with all possible combination of element of the Herbrand universe. However, the original meaning of the rules will be lost by their duplication.

For example, if the rule $A(X) \leftarrow B(X) \wedge C(X)$ have to be instantiated on the Herbrand universe a, b, c, three propositional rules will be generated ($A(a) \leftarrow B(a) \wedge C(a)$, $A(b) \leftarrow B(b) \wedge C(b)$ and $A(c) \leftarrow B(c) \wedge C(c)$).

However, suppose the case that the rule should actually be $A(x) \leftarrow B(x) \wedge \neg C(x)$ but the only counter examples are with a. The BACKPROPAGATION algorithm will correctly correct the rule on a, but it will not be able to correct the rules on b and c. Therefore, depending of the application, even with a finite Herbrand model, the KBAAN algorithm may not even be well fitted.

The second limitation is linked to the initial domain theory. If it is too badly defined, the algorithm might not be able to improve it and return a satisfying domain theory. Since the architecture of the artificial neural network only depend of the initial domain theory, an ANN based on a bad initial domain theory may unable

to explain a set of training example.

The third limitation to note is the inability for the output domain theory to deal with unknown facts. Because of the construction of the ANN, to give a meaning to the output nodes, all the input node have to given and be 0 or 1. Based on that, it is impossible to not define a propositional symbol for an output domain theory query. Concepts like default negation have to be applied in order to be able to ask the kind of queries that can be handled by the initial domain theory.

For example, suppose the following clause in a program $P : A \leftarrow B \wedge C$ with B's value known to be false and C's value unknown. If a SLD solver is asked to "evaluate" A, it will initially evaluate B (or C, depending of its policy. In this case reverse the reasoning). This sub goal will fails, and the algorithm will stop without trying to evaluate the other body's term C. If the same query is carried with a neural network generated by KBANN, C will have to be given (there is not convention to represent the unknown value). The simplest solution would be therefore to run two time the neural network with a value of 0 and 1 for C, and finally to check if the outputs are the same i.e. if C has an influence on the result.

Therefore, if this technique is generalized, if N terms are unknown, the neural network will have to be run 2^N times. When, in the best case, the SLD solver will only need to be run once.

It is also important to note that the artificial neural network generated by the KBANN algorithm does not compute the immediate consequences like the ones generated by the method presented after. In fact, it only allow to specify the true/false value of terms that are not defined in the head of a Horn clause, and to evaluate the value of the other terms.

The last limitation, which is actually more a remark, is that the algorithm does not simplify the hypothesis. For example, if for a given domain theory that explain the training examples there is a simpler domain theory that also explains the training examples, KBANN will be unable to simplify the domain theory and it might produce an over-complex Artificial Neural Network. This problem seems not to be primordial for domain theory given by humans, but in the closed loop Artificial Neural Networks and Logic Programs generation/extraction, this incapability to generalize hypotheses might become a big gap. We will come back on this point later.

2.1.1.5 Conclusion

KBANN is an algorithm based on a very interesting idea, but its lacks make it not powerful enough to be able to deal with a lot of the problems. However, it opened the promising field of the conjoint utilization of Artificial Neural Networks and Logic Programs for Machine Learning problem.

An important fact to note is the following one: A lot a literature presents a wrong version of the KBANN algorithm. For example, the KBANN algorithm presented in the book "Machine Learning" [**?**] in chapter 12 is not actually able to handle programs in which terms are defined in more than one head of clause.

For example, the following program is impossible to be computed with this KBANN algorithm:

$$P = \left\{ \begin{array}{l} p \leftarrow q \\ p \leftarrow r \end{array} \right\}$$

2.1.2 T_P Neural Network for Propositional Logic Programs

In the paper "Towards a massively parallel computational model for logic programming" [**?**], Steffen Holldobler and Yvonne Kalinke present prove by construction that for every definite propositional logic program

there exists a 3 layers recurrent neural network that built of binary threshold units that computes the immediate consequence operator.

In the paper "Logic Programs and Connectionist Networks" [**?**], Pascal Hitzler, Steffen Holldobler, Anthony Karel Seda prove the extension of this theorem to the case of normal (general) propositional logic programs.

This algorithm gives an elegant way to translate a propositional program into a neural network.
In the following section is presented the algorithm constructed by the second proof (the first algorithm is a special case of the second one) and a simple example of computation.

2.1.2.1 Algorithm

Algorithm 2 T_P Neural Network

1. For every propositional variable A of the program P add a input binary threshold unit u_A and a output binary threshold unit w_A with a threshold value of 0.5.
2. For each clause $A \leftarrow L_1, ..., L_n$ of P
   (a) Add a binary threshold unit c to the hidden layer.
   (b) Connect c to the unit representing A (w_A) in the output layer with weight 1.
   (c) For each literal L_j , $1 \leq j \leq k$, connect the unit representing L_j in the input layer (u_{L_j}) to c and, if L_j is an atom, then set the weight to 1 ; otherwise set the weight to -1.
   (d) Set the threshold c to $l - 0.5$, where l is the number of positive literals occurring in $L_1, ..., L_k$.

In two steps the artificial neural network computes the immediate consequence of the interpretation presented in input with the following convention:

To present the interpretation I to the neural network, set the u_A input node value to 1 if $A \in I$, otherwise set the input node value to 0.

The computed output interpretation I' is presented with the following convention:
For all output node w_A of the neural network. $A \in I'$ if the value of w_A is 1, otherwise $A \notin I'$.

2.1.2.2 An example

Here is a simple example of generation of run of this algorithm.

Suppose the following logic program:

$$P = \left\{ \begin{array}{l} C \leftarrow A, B \\ A \leftarrow \neg C \\ B \leftarrow A \end{array} \right\}$$

The neural network generated by the algorithm will be the following one:

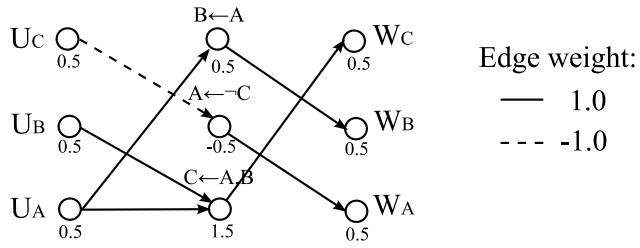


Figure 2.3: example of generated neural network

Here is an example of immediate operator computation, how the network compute $T_P(\{A, B, C\})$:

$$T_P(\{A,B,C\})? \Rightarrow \left\{ \begin{array}{l} U_A = 1 \\ U_B = 1 \\ U_C = 1 \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} B \leftarrow A = 1 \\ A \leftarrow \neg C = 0 \\ \neg C \leftarrow A, B = 1 \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} W_A = 0 \\ W_B = 1 \\ W_C = 1 \end{array} \right\} \Rightarrow \{B, C\}$$

2.1.2.3 The Core Method

The Core Method was first studied by Steffen Holldobler and Yvonne Kalinke [**?**]. The idea is basically to use the T_P neural network generation technique previously presented and to connect its output nodes to its corresponding input nodes. The recursive resulting network may compute iteratively a fix point of the represented T_P operator. Moreover, standard connectionist training techniques can be used to train the network and therefore construct a T_P operator based on sets of examples.

To use such techniques, a binary threshold unit is not well fitted. However, it has been shows that bipolar sigmoidal units based neural network can be trained to represent immediate consequence operator.

2.1.2.4 Remarks and conclusion

This technique gives a sound and complete method to generate neural networks that compute immediate consequence operator based on propositional logic programs. More than that, the generated networks can be used with other techniques like learning or extracting (presented after).

However, it is important to note that, generally, knowledge can't be represented with only propositional logic programs. There are problems that can't actually be solve by these techniques.

2.2 Propositional extraction from trained ANN

Once a network is built from one with one of the previous algorithms, and trained on a set of examples, extraction algorithms extract the encoded knowledge into a set of logic rules.

The following paragraph presents three techniques that extract propositional logic programs extraction from artificial neural networks. The first technique works by enumerating the elements of the Herbrand Base of the final program, and deducing a set of clauses with a similar immediate consequence operator. The second and the third techniques construct a set of rules based on the actual graph architecture of the neural network.

2.2.1 Herbrand Base Exploration techniques

Jens Lehmann, Sebastian Bader and Pascal Hitzler, submitted in 2005 in the paper "Extracting reduced logic programs from artificial neural networks" [**?**] three interesting techniques to extract propositional logic

programs from artificial neural networks. Contrary to the techniques presented later, these techniques do not work by analyzing the architecture of the network, but by running it on a set of inputs. Therefore the neural network is simply used as a T_P operator (consequences operator) black box, and the convention on values (how to encode *true*, $false$ and *unknown*) to use it do not have any importance i.e. given a mapping f, the algorithm return P with $T_P = f$.

The first presented technique works by enumerating all possible inputs of the T_P operator (2^n, with n the number of terms) encoded in the neural network, and constructing a sound and complete logic definite program with a similar T_P operator.

The second presented technique works by enumerating all possible inputs of the T_P operator (2^n, with n the number of terms) encoded in the neural network, and constructs a definite sound, complete normal logic program with a similar T_P operator.

Those two techniques are computationally extremely expensive but the returned programs are guaranteed to be minimal.

Those two algorithms are extremely simple, but extremely slow. Therefore, a more efficient technique is presented. However the extracted program is not anymore guaranteed to be minimal. This last technique is said to be greedy but it seems to be far from obvious.

Here is presented the second and the third technique. The second is interesting to understand the underlying idea, while the third one is a potentially interesting solution.

2.2.1.1 The second technique: Normal Full Exploration and Reduced program

Algorithm 3 Full Exploration [**?**]

1. Let T_P be a consequence operator given as input mapping ($T_P : I_P \Rightarrow I_P$).
2. Let B_P the set of all terms.
3. Initialize $P = \emptyset$.
4. For every interpretation $I = r_1, ..., r_a \in I_P$
5. $s_1, ..., s_b = B_P / I$.
6. For each element $h \in T_P(I)$
7. add a clause $h \leftarrow r_1, ..., r_a, \neg s_1, ..., \neg s_b$ to P.
8. Repeat the following loop as long as possible
   (a) If there are two clauses $C_1 = p \leftarrow q, r_1, ..., r_a, \neg s_1, ..., \neg s_b$ and $C_2 = p \leftarrow \neg q, t_1, ..., t_c, \neg u_1, ..., \neg u_d$ with $C_1 \neq C_2$, $\{r_1, ..., r_a\} \subseteq \{t_1, ..., t_c\}$ and $\{s_1, ..., s_b\} \subseteq \{u_1, ..., u_d\}$, then remove $\neg q$ in the body of C_2.
   (b) If there are two clauses $C_1 = p \leftarrow \neg q, r_1, ..., r_a, \neg s_1, ..., \neg s_b$ and $C_2 = p \leftarrow q, t_1, ..., t_c, \neg u_1, ..., \neg u_d$ with $C_1 \neq C_2$, $\{r_1, ..., r_a\} \subseteq \{t_1, ..., t_c\}$ and $\{s_1, ..., s_b\} \subseteq \{u_1, ..., u_d\}$, then remove q in the body of C_2.
   (c) If there are clauses C_1 and C_2 with $C_1 \neq C_2$ in P and C_1 subsumes C_2, then remove C_2.
   (d) If a literal appears twice in the body of a clause, then remove one occurrence.
   (e) If a literal and its negation appear in the body of a clause, then remove this clause.
9. Return P.

2.2.1.2 Example for the Full Exploration with Normal logic program technique

Suppose the following mapping f:

$$f = \left\{ \begin{array}{l} \{\emptyset\} \mapsto \{p\} \\ \{p\} \mapsto \{\emptyset\} \\ \{q\} \mapsto \{p\} \\ \{p,q\} \mapsto \{q,p\} \end{array} \right\}$$

When the algorithm is applied on the mapping f. The first time the step 8 is reached P is equal to:

$$P = \left\{ \begin{array}{l} p \leftarrow \neg p, \neg q \\ p \leftarrow \neg p, q \\ q \leftarrow p, q \\ p \leftarrow p, q \end{array} \right\}$$

After the reduction, the final output is:

$$P = \left\{ \begin{array}{l} p \leftarrow \neg p \\ p \leftarrow q \\ q \leftarrow p, q \end{array} \right\}$$

Remark 2.2. $p \leftarrow \neg p \wedge q$ and $p \leftarrow p \wedge q$ implies $p \leftarrow q$.

Remark 2.3. We have $T_P = f$ which is the expected result.

2.2.1.3 The third technique: Greedy Extraction Algorithm and Intelligent Program Search

Definition 2.3. [**?**] Let B_P be a set of predicates. The score of a clause $C : h \leftarrow B$ with respect to a program P is defined as

$$score(C, P) = |\{I | I \subseteq B_P \text{ and } h \notin T_P(I) \text{ and } I \models B\}|$$

The *score* is the number of support of B that does not implies h thought P.

Definition 2.4. $T_P^q(I) = \{q\}$ if $q \in T_P(I)$, and $T_P^q(I) = \emptyset$ otherwise

Definition 2.5. Let T_P be an immediate consequence operator, and h be a predicate. We call $B = p_1, ..., p_a, \neg q_1, ..., \neg q_b$ allowed with respect to h and T_P if the following two properties hold:

For every interpretation $I \subseteq B_P$ with $I \models B$ we have $h \in T_P(I)$.

There is no allowed body $B' = r_1, ..., r_c, \neg t_1, ..., \neg t_d$ for h and T_P with $B' \neq B$ such that $\{r_1, ..., r_c\} \subseteq \{p_1, ..., p_a\}$ and $\{t_1, ..., t_d\} \subseteq \{q_1, ..., q_b\}$.

Algorithm 4 Greedy Extraction Algorithm [**?**]

1. Let T_P and $B_P = \{q_1, ..., q_m\}$ be the input of the algorithm.
2. Initialize $Q = \emptyset$.
3. For each predicate $q_i \in B_P$:
   (a) construct the set S_i of allowed clause bodies for q_i
   (b) initialize: $Q_i = \emptyset$
   (c) repeat until $T_{Q_i} = T_P^{q_i}$:
      i. Determine a clause C of the form $h \leftarrow B$ with $B \in S_i$ with the highest score with respect to Q_i.
      ii. If several clauses have the highest score, then choose one with the smallest number of literals.
      iii. $Q_i = Q_i \cup C$
4. $Q = Q \cup Q_i$

2.2.1.4 Example for the Greedy technique

Suppose the following mapping f:

$$f = \left\{ \begin{array}{l} \{\emptyset\} \mapsto \{p\} \\ \{p\} \mapsto \{\emptyset\} \\ \{q\} \mapsto \{p\} \\ \{p, q\} \mapsto \{q, p\} \end{array} \right\}$$

The possible clause bodies for p (B_p) are i.e. the clauses with p in the head :

clause body	body allowed
$\emptyset$	no, $\emptyset \in \{p\}$ and $p \neq T_P(\{p\})$
p	no, $p \neq T_P(\{p\})$
q	yes
$\neg p$	yes
$\neg q$	no, $p \neq T_P(\{p\})$
p, q	no, q is smaller
$p, \neg q$	no, $p \neq T_P(\{p\})$
$\neg p, q$	no, q is smaller
$\neg p, \neg q$	no, $\neg p$ is smaller

And the algorithm builds the program in this way:

$$Q_p = \left\{ \begin{array}{l} p \leftarrow q \text{ (score : 2, } \{q\} \text{ and } \{p, q\} \text{)} \\ p \leftarrow \neg p \text{ (score : 1, } \{\neg p\}) \end{array} \right\} \; Q_q = \left\{ \; q \leftarrow p, q \; \right\}$$

Finally, the result is:

$$Q = Q_p \cup Q_q = \left\{ \begin{array}{l} p \leftarrow \neg p \\ p \leftarrow q \\ q \leftarrow p, q \end{array} \right\}$$

Remark 2.4. We have $T_P = f$ which is the expected result.

2.2.1.5 Remarks and conclusion

Those techniques are relatively simple. More of that, they do not even need as input an artificial neural network, but only an immediate operator mapping i.e. they are independent of the way to encode the actual knowledge, and therefore, they can actually be used with any other kind of representation as soon as the T_P operator mapping is computable.

Another extremely interesting feature is that these techniques are sound and complete in the respect of the input immediate consequence mapping ($T_P = f$ with P the result of the algorithms with f as input). In the other presented techniques, those properties do not hold and we need to choose between the soundness or the completeness.

However, this property can also be a drawback: it can be interesting not to extract a complete set of rules, but just a subset that explain "not too badly" the examples (through kind of measure significance) in order to do not extract a too large and maybe over fitted program.

This is a common feature in machine learning techniques in the case of noisy data. Nevertheless, informally we can imagine such extensions. For example, by associating a confidence measure to T_P query and filtering generated rules base on this information.

Another limitation is the following one; these techniques intrinsically work by going through all the possible interpretations i.e. the number of interpretation is exponentially proportional to the number of terms in the program (In the case of immediate consequence encoded in an artificial neural network, the number of interpretation is exponential in term of the number of input unit). Therefore, form this point of view, those algorithms are actually intractable and may be difficult to be used with large program instances.

2.2.2 Architectural analysis techniques

In the paper "Symbolic knowledge extraction from trained neural networks: A sound approach" [**?**], A.S. d'Avila Garcez, K. Broda and D.M. Gabbay present two knowledge extraction techniques based on the architecture of a T_P operator network (of the kind defined in the "Logic Programs and Connectionist Networks" paper).

In the following section is presented the algorithms and three examples of computation. This description of the algorithms is different from the one presented in the "Symbolic knowledge extraction from trained neural networks" papers. I actually tried to present them in a different, and, I think, more easily understandable way.

Both of the algorithms generate from the ANN, a set of propositional logic rules. Every node of the network is considered as a proposition that is true in the considered model if the neuron value is larger than a fixed parameter. The output rules express the activation condition of the output nodes, depending of the activation of the input nodes. In the case of the KBANN neural network or the T_P neural network, the extracted rules are directly the expected hypothesis to extract.

The difference between the two algorithms is the following one: The first algorithm is sound and complete, but it does not allow negative weighted neural network edges. On the opposite, the second algorithm is sound, it accept negative weighted neural network links and is more efficient than the first one. But it is not complete and it produces a lot of intermediate rules.

An example of incompleteness of the second algorithm is given after its description.

2.2.2.1 The sound and complete algorithm

Definition 2.6. Let N be a neural network with p input neurons $\{i_1, ..., i_p\}$, q output neurons $\{o_1, ..., o_q\}$ and $v \in \mathbb{R}^p$ an vector of input values. We define $o_i(v)$ to be the value of the output neuron o_i when the values v are presented to the input neurons of the network.

Definition 2.7. Let $u = \{u_1, ..., u_p\} \in \mathbb{R}^p$ and $v = \{v_1, ..., v_p\} \in \mathbb{R}^p$ be two vector of input values. u is said greater than v iff $\forall i, u_i > v_i$.

The first algorithm is based on the condition that all the network's weights are positive or null.

Theorem 2.2.1. *Suppose $I = \{i_1, ..., i_p\}$ the input neurons of an all weighted positive network, and o one of the output neuron of this network. Since the different possible values of the input neurons are known (For example, the possible values are $V = \{0, 1\}$ or $V = \{-1, 1\}$ in the network presented in the previous techniques). We know that for two different input vectors of values $v_1 \in V^p$ and $v_2 \in V^p$, if $v_1 \geq v_2$ then $o(v_1) \geq o(v_2)$.*

Two direct implications are the following ones. Let *Amin* the minimum value a neuron should reach to be considered as activated.

Corollary 2.2.2. *Let o be an output neuron and $u = \{u_1, ..., u_p\} \in \mathbb{R}^p$ and $v = \{v_1, ..., v_p\} \in \mathbb{R}^p$ be two vectors of input values with $u >= v$. If v activate o i.e. $o(v) \geq Amin$, then, u also activate o i.e. $o(u) \geq Amin$.*

Corollary 2.2.3. *Let o be an output neuron and $u = \{u_1, ..., u_p\} \in \mathbb{R}^p$ and $v = \{v_1, ..., v_p\} \in \mathbb{R}^p$ be two vectors of input values with $u >= v$. If u does not activate o i.e. $o(u) < Amin$, then, v also does not activate o i.e. $o(v) < Amin$.*

Based on those corollaries, the first algorithm searches for all output neurons, the vectors of input values that activate them. Thanks to the two previous corollaries, the research is optimized by not testing all the vectors of input.

The following graph shows an example of exploration from the bottom (minimum vector of input values) for a neuron with four different antecedent input neurons. In this bottom-top exploration, when a node activates the target neuron, a new rule is created and the branch closed. At the state of the algorithm represented by the graph, the created rules are $X_1 \rightarrow O$, $X_1 \vee X_2 \rightarrow O$ and $X_3 \rightarrow O$. Since $X_1 \rightarrow O$ subsumes $X_1 \vee X_2 \rightarrow O$, this last rule will be deleted.

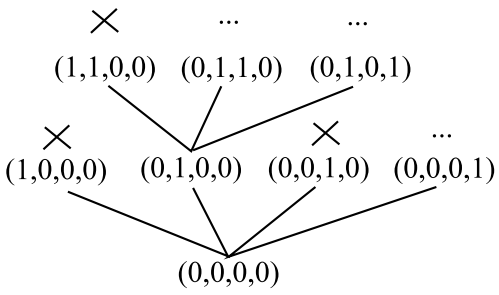


Figure 2.4: example of bottom-top exploration

Algorithm 5 Sound and complete extraction based on the architecture Part 1

Require: The artificial neural network does not have negative weight
Let $Amin$ be the minimum activation value
Let $infB$ be the minimum value of the input neurons
Let $supB$ be the maximum value of the input neurons
Let inf and sup be a set of couple of input vector and depth
for all ouput neuron o **do**
Compute $neuron$, the set of all the input neurons connected to o (even indirectly)
Set $size$ the number of element of $neuron$
Let top be the input vector for $neuron$ filled with supB values
Let $bottom$ be the input vector for $neuron$ filled with supB values
if top does not activate o **then**
Continue
else if $bottom$ does activate o **then**
Add the rule $\rightarrow o$
Continue
else
Add top to sup and $bottom$ to inf with depth $size$ and 0
while sup and inf are not empty **do**
if inf is not empty **then**
Take and **remove** one element e from inf
if the depth of e under $size/2 - 1$ **then**
for $i \in 0, size - 1$ with $e[i] \neq supB$ **do**
Set $patern = e$ with in addition $patern[i] = supB$
if $patern$ activates o **then**
Add the rule $\bigwedge\{neuron[i] | patern[i] = supB\} \rightarrow o$
else
Add $patern$ to inf with a depth equal to the depth of e plus one
end if
end for
end if
end if
if sup is not empty **then**
Take and **remove** one element e from sup
if the depth of e over $size/2 + 1$ **then**
for $i \in 0, size - 1$ with $e[i] \neq infB$ **do**
Set $patern = e$ with in addition $patern[i] = infB$
if $patern$ activates o **then**
Add the rule $\bigwedge\{neuron[j] | patern[j] = supB\} \rightarrow o$
Add $patern$ to sup with a depth equal to the depth of e minus one
end if
end for
end if
end if

Algorithm 6 Sound and complete extraction based on the architecture Part 2

```
        end while
    end if
end for
```
Removes the rules that are subsumed my other rules
Removes the rules that are inconsistent
Removes the rules of the shape "...→⊥"

2.2.2.2 The sound algorithm

Contrary to the previous algorithm, this algorithm allows to have negative weighted network.

Definition 2.8. Let N be a neural network with p input neurons $i_1, ..., i_p$, r hidden neurons $n_1, ..., n_r$ and q output neurons $o_1, ..., o_q$. A sub network N_0 of N is a Basic Neural Structure (BNS) iff either N_0 contains exactly p input neurons, 1 hidden neuron and 0 output neurons of N, or N_0 contains exactly 0 input neurons, r hidden neurons and 1 output neuron of N

Definition 2.9. A single hidden layer neural network is said to be regular if its connections from the hidden layer to each output neuron have either all positive or all negative weights.

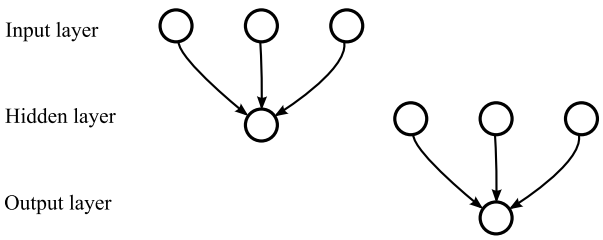


Figure 2.5: Two example of BNS networks

The root idea of this algorithm is to divide an actual neural network into several regular BNS sub neural network (one for each non input neuron) thanks to the **Transformation algorithm** and **BNS division algorithm** and to generate for each of them a set of rules with the firstly presented algorithm. Those logical rules are defined in order to keep the sum of them sound in respect of the T_P mapping computed by the neural network. However, because of the decomposition, the completeness is not guaranteed anymore.

Algorithm 7 Transformation Algorithm [**?**]

Require: The artificial neural network is a non recursive, two layer network with one output neuron
for Neuron n of the input layer connected with an negative weight to the ouput neuron **do**
Add the neuron Not_n to the input layer, and connect it to the output neuron for every negative connection of n. The weight of this connection is the opposite (addition) of the weight of the connection of n.
Remove all the negative weighted connections to n
end for

Algorithm 8 Split neural network N into BNSs [**?**]

Let N be the input neural network
for Neuron n of the network that are not input neuron **do**
Create the sub BNS network b of N that only contains the neuron n and the neurons directly connected to n.
end for

Algorithm 9 Sound extraction based on the architecture

Let N be the input neural network
Divide the neural network N into several BNS (**Split neural network N into BNSs**)
Transform the new BNSs into regular BNSs (**Transformation Algorithm**)
for $n \in BNS$ **do**
Apply the **Sound and complete extraction based on the architecture for BNS** algorithm. Depending on the type of the neuron n, the upper bound will be adapted.
end for
Removes the rules that are subsumed my other rules
Removes the rules that are inconsistent
Removes the rules of the shape "... $\rightarrow \bot$"

The **Sound and complete extraction based on the architecture for BNS** algorithm is an evolution of the **Sound and complete extraction based on the architecture** algorithm in the case the input network have only two layers and one output neuron. With this property, the input neurons are sorted according to theirs associated weights, and the exploration optimized based on the two following theorems.

Theorem 2.2.4. *Suppose $I = \{i_1, ..., i_p\}$ the input neurons of a BNS network with an increasing activation function, o the output neuron of this network, and $W = \{w_1, ..., w_p\}$ the different weights.Suppose the weights are sorted i.e. $i < j$ iff $w_i \geq w_j$.Suppose a vector of values $u \in V^p$.*

If v_1 be a vector of values based on u with the j_1^{th} component initially to the lower bound, sets to the input upper bound value, activates the output neuron. Therefor for all $j_2 < j_1$, v_2, the vector of values based on u with the j_2^{th} component initially to the lower bound, sets to the input upper bound value also activates the output neuron.

Theorem 2.2.5. *Suppose $I = \{i_1, ..., i_p\}$ the input neurons of a BNS network with an increasing activation function, o the output neuron of this network, and $W = \{w_1, ..., w_p\}$ the different weights.Suppose the weights are sorted i.e. $i < j$ iff $w_i \geq w_j$.Suppose a vector of values $u \in V^p$.*

If v_1 be a vector of values based on u with the j_1^{th} component initially to the lower bound, sets to the input upper bound value, does not activate the output neuron. Therefor for all $j_2 > j_1$, v_2, the vector of values based on u with the j_2^{th} component initially to the lower bound, sets to the input upper bound value also does not activate the output neuron.

Here is a simple example of exploration from the bottom.

Since the node $(0, 0, 1, 0)$ activates the output neuron. The nodes $(0, 1, 0, 0)$ and $(1, 0, 0, 0)$ also active the output neuron, and the inferred rule is $1(x_1, x_2, x_3) \rightarrow c$.

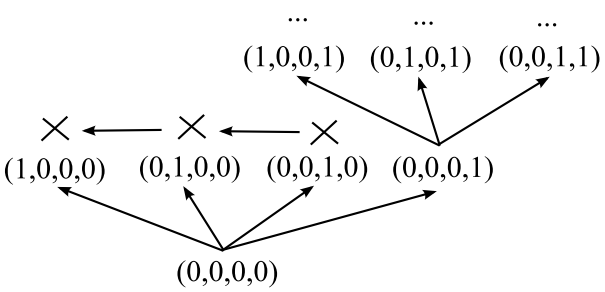


Figure 2.6: example of bottom-top optimized exploration

Algorithm 10 Sound and complete extraction based on the architecture for BNS Part 1

Require: The artificial neural network does not have negative weight

Let $Amin$ be the minimum activation value
Let $infB$ be the minimum value of the input neurons
Let $supB$ be the maximum value of the input neurons
Let inf and sup be a set of couple of input vector and depth
Compute $neuron$, the set of all the input neurons connected to o (even indirectly)
Set $size$ the number of element of $neuron$
Let top be the input vector for $neuron$ filled with supB values
Let $bottom$ be the input vector for $neuron$ filled with supB values
Let c be the output neuron
if top does not activate o **then**
    Continue
else if $bottom$ does activate o **then**
    Add the rule $\rightarrow o$
    Continue
else
    Add top to sup and $bottom$ to inf with depth $size$ and 0
    while sup and inf are not empty **do**
        if inf is not empty **then**
            Take and **remove** one element e from inf
            if the depth of e under $size/2 - 1$ **then**
                for $i \in 0, size - 1$ with $e[i] \neq supB$ **do**
                    Set $patern = e$ with in addition $patern[i] = supB$
                    if $patern$ activates o **then**
                        Add the rule $1(\{neuron[j]|e[j] = infB, j \in [1, i-1]\}) \wedge \{neuron[j]|e[j] = supB\} \rightarrow o$
                        Break
                    else
                        Add $patern$ to inf with a depth equal to the depth of e plus one
                    end if
                end for
            end if
        end if
        if sup is not empty **then**
            Take and **remove** one element e from sup
            if the depth of e over $size/2 + 1$ **then**
                for $i \in 0, size - 1$ with $e[i] \neq infB$ **do**
                    Set $patern = e$ with in addition $patern[i] = infB$
                    if $patern$ activates o **then**
                        Add $patern$ to sup with a depth equal to the depth of e minus one
                    else
                        Break

Algorithm 11 Sound and complete extraction based on the architecture for BNS Par 2

end if
end for
Add the rule $(\sum_{j=i+1}^{size} e[j] == supB)(\{neuron[j]|e[j] = supB, j \in [i+1, size]\}) \wedge \{neuron[j]|e[j] = supB, j \in [1, i]\} \rightarrow o$
end if
end if
end while
end if

2.2.2.3 Example 1 : First algorithm

This example shows the work of the first algorithm.

We generate a network with the architecture described in the General Consequence operator part based on the clauses B, which imply the rules R.

$$B = \left\{ \begin{array}{l} A \rightarrow A \\ B \rightarrow B \\ C \rightarrow C \end{array} \right\}$$

$$R = \left\{ \begin{array}{ll} A \rightarrow A & \neg A \rightarrow \neg A \\ A \wedge \neg A \rightarrow \bot & \top \rightarrow A \vee \neg A \\ B \rightarrow B & \neg B \rightarrow \neg B \\ B \wedge \neg B \rightarrow \bot & \top \rightarrow B \vee \neg B \\ C \rightarrow C & \neg C \rightarrow \neg C \\ C \wedge \neg C \rightarrow \bot & \top \rightarrow C \vee \neg C \end{array} \right\}$$

Next to that, the network is trained on a set of example representing the rules:

$$B'' = \left\{ \begin{array}{l} A \rightarrow A \\ B \rightarrow B \\ C \rightarrow C \\ A \wedge B \rightarrow C \\ A \wedge \neg B \rightarrow \neg C \end{array} \right\}$$

More precisely, the training examples are:

Input	Expected Output
$\emptyset$	$\emptyset$
A	A
$\neg A$	$\neg A$
B	B
A, B	A, B, C
$\neg A, B$	$\neg A, B$
$\neg B$	$\neg B$
$A, \neg B$	$A, \neg B, \neg C$
$\neg A, \neg B$	$\neg A, \neg B$

The trained artificial neural network has the following shape:

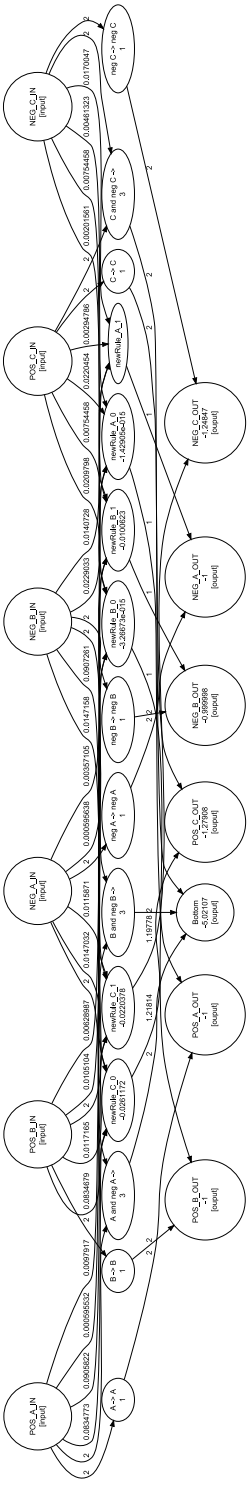


Figure 2.7: Trained artificial neural network

By applying the implementation of the algorithm, the extracted rules are :

```
A -> A
neg A -> neg A
B -> B
neg B -> neg B
```

```
C -> C
A and B -> C
neg C -> neg C
A and neg B -> neg C
```

They are exactly the rules we have been expected.

2.2.2.4 Example 2 : Second algorithm

This example shows the work of the second algorithm.

The input is given to be the neural network generated in the example of T_P Neural Network for Propositional Logic Programs.
The input neural network is the following one:

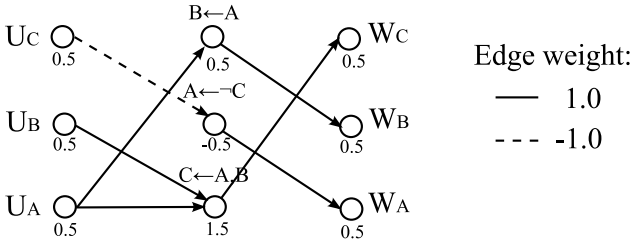


Figure 2.8: Input of the algorithm

It can be divided into six BNS sub neural networks:

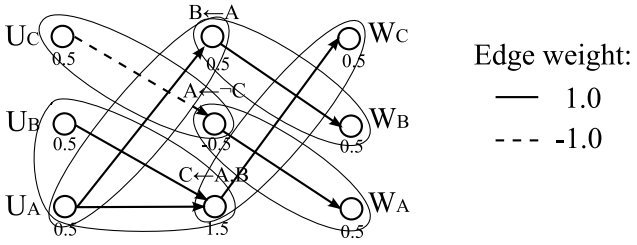


Figure 2.9: BNS division

And be regularized in the following one (note the change to U_C to $U_{\neg C}$):

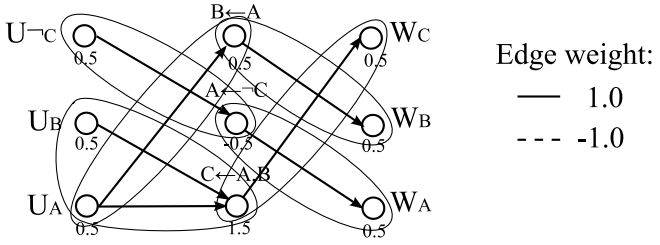


Figure 2.10: Regularized BNS division

For every BNS, a rule is extracted:

$$P' = \left\{ \begin{array}{l} 1(\neg C) \rightarrow \text{“}A \leftarrow \neg C\text{”} \\ 1(A) \rightarrow \text{“}B \leftarrow A\text{”} \\ 2(A,B) \rightarrow \text{“}C \leftarrow A,B\text{”} \\ 1(\text{“}A \leftarrow \neg C\text{”}) \rightarrow A \\ 1(\text{“}B \leftarrow A\text{”}) \rightarrow B \\ 1(\text{“}C \leftarrow A,B\text{”}) \rightarrow C \end{array} \right\}$$

Note that “$A \leftarrow \neg C$” represent a node of the neural network and not a logical equation.

With the following signification:

$n(a_1, ..., a_m)$ hold if and only if at least n of the a_i terms are evaluated to the true value.

Example 2.1. $2(a, b, c) \rightarrow d$ is equivalent to $((a \wedge b) \vee (b \wedge c) \vee (a \wedge c)) \rightarrow d$.

In this example, the neural network presented as input of the algorithm was generated by the T_P Neural Network generator algorithm based on the following rules:

$$P = \left\{ \begin{array}{l} C \leftarrow A,B \\ A \leftarrow \neg C \\ B \leftarrow A \end{array} \right\}$$

The extracted program P' is more complex that the initial program P, but its T_P operator is what was expected:

$$\forall I, T_{P'}(I) \cap \{A, B, C\} \subseteq T_P(I)$$

Actually, in this example, the extraction is sound and complete:

$$\forall I, T_{P'}(I) \cap \{A, B, C\} = T_P(I)$$

2.2.2.5 Example 3 : Second algorithm, case of incompleteness

This example shows the work of the second algorithm in the case of an incomplete extraction.

We apply the second algorithm on the same network used in the first example.

By applying the implementation of the algorithm, the extracted rules are:

```
1("A -> A") -> A
1("neg A -> neg A") -> neg A
1("B -> B") -> B
1("neg B -> neg B") -> neg B
1("C -> C") -> C
1("neg C -> neg C") -> neg C
1(A) -> "A -> A"
1(neg A) -> "neg A -> neg A"
2(A and neg A) -> "A and neg A -> "
1(B) -> "B -> B"
1(neg B) -> "neg B -> neg B"
2(B and neg B) -> "B and neg B -> "
```

```
1(C) -> "C -> C"
1(neg C) -> "neg C -> neg C"
2(C and neg C) -> "C and neg C -> "
4(neg A and neg B and B and A) -> "newRule_C_0"
3(neg B and B and A) -> "newRule_C_0"
3(neg A and B and A) -> "newRule_C_0"
2(B and A) -> "newRule_C_0"
4(neg A and B and A and neg B) -> "newRule_C_1"
3(B and A and neg B) -> "newRule_C_1"
3(neg A and A and neg B) -> "newRule_C_1"
2(A and neg B) -> "newRule_C_1"
```

We get effectively the rule $2(B, A) \rightarrow$ `newRule_C_0`, but the rule $2($`newRule_C_0,C -> C`$) \rightarrow C$ that would complete the implication about $A \wedge B \rightarrow C$, is missing. Actually, we just have the part $1($`C -> C`$) \rightarrow C$ that represents the fact $C \rightarrow C$.

By changing the parameter *Amin* we can, for this example, producing the rule $2($`newRule_C_0,C -> C`$) \rightarrow C$ but we can also loosing $2(B, A) \rightarrow$ `newRule_C_0`.

This example illustrates the incompleteness of this method.

2.2.2.6 Remarks and conclusion

In the case of regular neural network, this extraction algorithm is sound and complete.

In the case of non-regular neural network, the algorithm is sound but not complete. However, the soundness can be traded with the completeness through a small modification of the algorithm.

The remark made about the Exploration techniques based algorithm hold here: Depending of the problem and the way the output of this algorithm need to be used, the generated program can be over fitted i.e. a sub set of the clauses can also explains "correctly" the examples (Basic machine learning problem).

By combining the example of the T_P Neural Network generator algorithm and the second example of this technique (sound extraction based on the network architecture), we notice that the final program P' (6 clauses, 6 terms) is bigger than the initial program P (3 clauses, 3 terms).

If the second algorithms is applied several times in a loop with an network generating algorithm, the generated program's size and the generated neural network's size will grow and will become harder and harder to use. Refining the rules at the end of every run of the extraction algorithm should be used in order to tackle the problem.

Chapter 3

Induction on Predicate Logic Programs through Artificial Neural Network

This chapter presents three techniques (technique 1, technique 2 and technique 3) for **inductively** learning **predicate logic programs** based on **artificial neural network** learning i.e. extraction of general rules based on a set of examples. These techniques are studied in sequence. The understanding of the lack of a given technique leads to the development of the next one. In this way, all the techniques have some common features.

The next example presents the work done by these technique.

The techniques will extract the rule that explains a set of predicate formulas. For example, given the next formulas. The techniques will train an artificial neural network equivalent to a consequence operator T_P with $P = \text{“}P(f(X)) \wedge Q(X) \Rightarrow R(g(X))''$.

$$\begin{aligned} P(f(a)) \wedge Q(a) \wedge Q(b) &\Rightarrow R(g(a)) \\ P(f(b)) \wedge Q(a) \wedge Q(b) &\Rightarrow R(g(b)) \\ P(f(a)) \wedge Q(b) \wedge Q(c) &\Rightarrow \emptyset \\ P(b) \wedge Q(b) &\Rightarrow \emptyset \end{aligned}$$

To use this technique we use a set of training example (association between sets of atoms) and some initial restrictions on this kind of rule we expect to learn. The algorithm will build and train an ANN able to simulate a immediate consequence operator, that correctly behave for the training examples, and able to deal with new examples with the general extracted knowledge.

The basic use of those techniques is captured in the following sequence of steps:

1. Choose restrictions on the rules the system will learn (called setting the language *bias*)
2. Construct a special artificial neural network using the language bias
3. Encode in the network the initial background knowledge (which can be empty)
4. Train the ANN on a set of examples encoded using the input convention defined on section .
5. Evaluate the learned rules on set of tests
   If the evaluation or the training fails, start again using a weaker language bias.
6. Extract the learned rules

Since it is possible to give the initial background knowledge and to directly run the evaluation step, each of those techniques can be used simply as a mapping operator of predicate atoms.

Predicate logic programming is a more powerful means of expression than propositional logic programming. However, because the general case allow an infinite Herbrand universe, the basic idea of the previously presented techniques cannot be adapted here.

The first presented technique (technique 1) is a general inductive technique that suffers convergence problems of the artificial neural network, because of its architectural complexity.

The second technique (technique 2) is an improvement and simplification of the first technique. However, if part of the input data are irrelevant for the problem, some convergence problems appear.

The third technique (technique 3) solves this problem. Although, the complexity of the produced network is bigger but training is far faster.

For each of those architectures, extraction techniques have been studied.

More precisely, those techniques learn a consequence operator based on a set of examples (mapping of a set of atoms to a set of atoms).

Definition 3.1. In this context, an *example* is a set of input predicate formulas and a set of expected output formulas.

The learned consequence operator is defined by a set of rules called the language bias. Restrictions on this set of rules are defined in the initial artificial neural network architecture. The more liberal those restrictions are, the more powerful the rules that can be learned, but the longer the learning process. A set of background knowledge rule can be given to the system to help the training. If there are not correct, they will be corrected during the learning process.

Contrary to most of the techniques of translation and induction of predicate logic programs through ANNs, our techniques are based on a strict analysis of ANN architecture and behaviour. At the construction of the network, it is necessary to define precisely the kind of rule the system should be able to infer. The feature has the great advantage to help the system to focus on important information. A typical example would be an induction analysis on a set of indexed example. We may not want (and this is true most of the time) the learned rules to take into consideration the index of the example, and therefore avoid over fitted solutions like the one that is to learn for every training example the correct answer only base on the index of the example.

From this ability to restrict the language bias, we expect to reduce the algorithmic cost of the learning (small research space), and to allow systems based on theses technique to be trained without fear of over fitting on very small example sets.

Example 3.1. Suppose that we do not allow ground terms in the inferred rules. The following training example will infer the rule $P(X) \Rightarrow Q(X)$, and not the rule $P(a) \Rightarrow R(a)$

$$P(a) \Rightarrow Q(a)$$
$$\emptyset \Rightarrow \emptyset$$

Generally, not allowing ground terms helps the system to generalise, and allow training of very small training example sets.

This chapter presents the common pre-required notions and conventions to understand the induction techniques described in the next chapters: The *Deduction rules* define in a intuitive way rules based on the form $Conditions \Rightarrow Implications$. The *term encoding* part presents the convention used to represent any logic term and more generally any list as a number. The *rule rewriting* defines a normal writing form for deduction.

Definition 3.2. A deduction rule, or rule, is a conjunctive predicate formula called body, and an atom called head. The semantic is defined as follow: When the body formula is evaluated as true, the rule fires, and the head atom is taken in consideration (produced). The allowed literals in the body and the head are restricted by the language bias.

3.1 Deduction rules

Example 3.2. There are four common examples of rules.

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

Definition 3.3. The depth of a term is defined as follow:

$$depth(X) = \begin{cases} 1 & \text{if } X \text{ is a constant } c \\ max(depth(Y_1),...,depth(Y_n)) & \text{if } X \text{ is a function with } X = f(Y_1,...,Y_n), f \in F \end{cases}$$

For the initial presentation, the language bias is a subset of the following rules.

1. The maximum number of atom in the body a rule
2. The maximum number of atoms with the same predicate in the body of a rule
3. The maximum number of arguments for any predicate
4. The maximum depth of a term
5. The term in the head must occur in the body of the rule
6. Ground terms are allowed in the head or the body

Technique 1 has predefined restriction 1, 3 and 4. The technique 2 has predefined restriction 2 and 3.

These restrictions are in general not a problem because they are changeable i.e. if a learning fail because the restrictions are too strong, the user can relax them and star again the learning process.

3.2 Term encoding

In the network, every term will be represented by an integer. The solution explored here is based on Cantor diagonalization ideas and the fact that $\mathbb{N} \in \aleph_0$. In the following lines is presented the function $encode : T \rightarrow \mathbb{N}$ that associates a unique integer for every term.
This encoding presents the following advantage. Using very simple arithmetic operations on integers, it is possible to define composition and extraction term operations on the number representing a term. For example, from the number n that represent 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 and the second argument $g(b, f(c))$, using only simple

arithmetic operations.

Assume that $Index : T \to \mathbb{N}$ is a function that gives a unique index to every function and constant.

Example 3.3.

$$Index(a) = 1$$
$$Index(b) = 2$$
$$Index(c) = 3$$
$$Index(f) = 4$$

Then define $Index' : T \to \mathbb{N}^+$ to be the function that recursively gives a list of unique indices for every term T as follow.

$$Index'(T) = \begin{cases} (Index(T), 0) & \text{if } T \text{ is a constant} \\ (Index(f), Index'(x_0), ..., Index'(x_n), 0) & \text{if } T \text{ is a function with } T = f(x_0, ..., x_n) \end{cases}$$

Example 3.4.

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

Now we define $E : \mathbb{N}^2 \to \mathbb{N}$ to be a bijective function from $\mathbb{N}^2$ to $\mathbb{N}$, and $D : \mathbb{N} \to \mathbb{N}^2$ the inverse function.

E is based on the following indexing of $\mathbb{N}^2$.

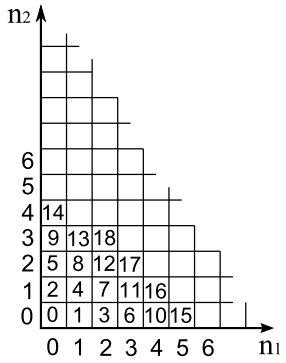


Figure 3.1: $\mathbb{N}^2$ to $\mathbb{N}$ correspondence

The mathematical definition of E and D are the following one:

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

$$D(n) = \begin{cases} p & = & \left\lfloor \frac{\sqrt{1+8.n}-1}{2} \right\rfloor \\ n_2 & = & n - \frac{p.(p+1)}{2} \\ n_1 & = & p - n_2 \end{cases}$$

With p an intermediate variable.

Let's define the two components of D as D_1 and D_2 i.e. $D(X) = (D_1(X), D_2(X))$.

Proof. Let's define $p = n_1 + n_2$.

$$\begin{aligned} n &= \frac{p \cdot (p+1)}{2} + n_2 \\ n' &= \frac{p \cdot (p+1)}{2} \\ n'' &= \frac{(p+1) \cdot (p+2)}{2} \end{aligned}$$

With n' the smallest value in the diagonal that contains n, and n'' the the smallest value in the diagonal next to the one that contains n.

$$n' \leq n < n''$$

$p \in \mathbb{N}$ then

$$\begin{aligned} p &= \frac{\sqrt{1+8.n'}-1}{2} = \left\lfloor \frac{\sqrt{1+8.n}-1}{2} \right\rfloor \\ p+1 &= \frac{\sqrt{1+8.n''}-1}{2} \\ n_2 &= n - \frac{p.(p+1)}{2} \\ n_1 &= p - n_2 \end{aligned}$$

Since $\frac{\sqrt{1+8.n'}-1}{2} \leq \frac{\sqrt{1+8.n}-1}{2} < \frac{\sqrt{1+8.n''}-1}{2}$, $f : x \mapsto \frac{\sqrt{1+8.x}-1}{2}$ is a strictly increasing function, and $f(\mathbb{N}) = \mathbb{N}$.

$$p = \left\lfloor \frac{\sqrt{1+8.n}-1}{2} \right\rfloor$$

□

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

$$E'([n_1, ..., n_m]) = \begin{cases} n_1 & \text{if } m = 1 \\ E'([n_1, ..., n_{m-2}] . E(n_{m-1}, n_m)) & \text{if } m > 1 \end{cases}$$

$$D'(X) = \begin{cases} [0] & \text{if } X = 0 \\ [D_1(X)] . D'(D_2(X)) & \text{if } m \geq 0 \end{cases}$$

Let's define E'' and D'' be the recursive extensions of E' and D' in $\mathbb{N}^+$.

$$E''([n_1, ..., n_m]) = \begin{cases} n_1 & \text{if } m = 1 \\ E'(E''(n_1), ..., E''(n_m)) & \text{if } m > 1 \end{cases}$$

$$D''(X) = [D''(x_1), ..., D''(x_n)] \text{ with } [x_1, ..., x_n] = D'(X)$$

We finally define $encode : T \to \mathbb{N}$ in the following way:

$$encode(T) = E''(Index'(T))$$

With this encoding, every term will have an unique associated integer number.

Example 3.5. Here is an example of encoding the term $f(a, b)$ with $index(a) = 1$, $index(b) = 2$ and $index(f) = 3$:

$$\begin{aligned} encode(f(a,b)) &= E''([4,[1,0],[2,0],0]) \\ &= E''([4,E(1,0),E(2,0),0]) \\ &= E''([4,1,3,0]) \\ &= E''([4,1,3,0]) \\ &= E'([4,1,3,0]) \\ &= E(4,E(1,E(3,0))) \\ &= E(4,E(1,6)) \\ &= E(4,34) \\ &= 775 \end{aligned}$$

Example 3.6. Here is an example of encoding the term $f(a, f(b, c))$:

$$\begin{aligned} encode(f(a,f(b,c))) &= E''([4,[1,0],[4,[2,0],[3,0],0],0]) \\ &= E''([4,E(1,0),[4,E(2,0),E(3,0),0],0]) \\ &= E''([4,E(1,0),E(4,E(E(2,0),E(E(3,0),0))),0]) \\ &= E(4,E(E(1,0),E(E(4,E(E(2,0),E(E(3,0),0))),0))) \\ &= 508603740163451130603064765127108648 \approx 5.10^{35} \end{aligned}$$

The figure 3.2 gives a graphical representation of the recursive list.

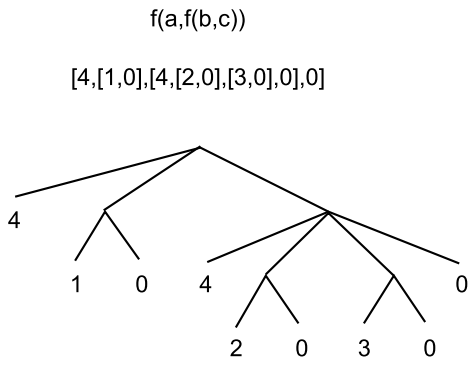


Figure 3.2: $\mathbb{N}^2$ to $\mathbb{N}$ correspondence

This encoding is based on the recursive syntactic representation of terms, and is used in this technique to present the input and read the output terms. The same procedure may also be used to represent and deal with any other list (lists of symbols like list of integers ($[1, 2, 3]$) or list of characters ($[a, b, c]$), or even lists of lists ($[[1, 2, 3], b, c]$)) inside the network. This last point is developed later.

This encoding allows a very simple extraction of sub-terms are or more generally, sub-elements of the list. For example, from a integer that represent the term $f(a, g(b))$, it is very simple to extract the index of f and the integers that represent a and $g(b)$:

Suppose E, D_1 and D_2 be 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 (E needs addition and multiplication operations. D_1 and D_2 need the square root, addition and multiplication operations).

The encoding integer of the i^{th} component of the list that is encoded as an integer N is equal to $D_1(D_2^i(N))$. If the list is empty, of if the i^{th} component does not exist $D_1(D_2^i(N)) = 0$. Zero is the last symbol of every list and the symbol returned as empty list or non existing element.

In the rest of this paper, an extraction neuron with the activation function E^i extract the i^{th} component of a list according to the previous equation. E^i can be constructed with several simple neurons as described in the figure 3.3. And a neuron with the activation function E, encode two term according to the function E defined below.

In term of terms, we have therefore $E^i(encode(f(x_1, ..., x_n))) = encode(x_i)$ if $i > 0$ and $E^0(encode(f(x_1, ..., x_n)))$ $= index(f)$ otherwise.

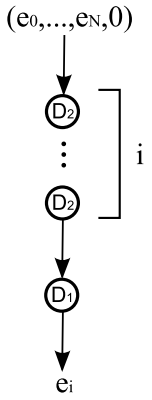


Figure 3.3: Extraction neurons

Example 3.7. The following example shows how we can do head and tail extraction operations of lists of symbols with only D_1 and D_2.

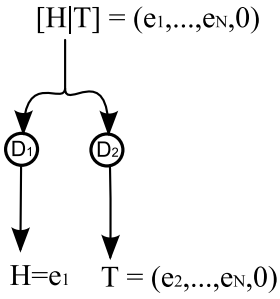


Figure 3.4: Extraction of the head and th tail of a list

3.3 Equality

Equality is one of the basic notions of this technique. Checking equality can be achieved by *equality neurons* based on Gaussian or diff arc tangent activation functions.

In the technique 2, equality tests are made over terms, and the back propagation does not follow through equality neurons i.e. equality is just an evaluating test. In the technique 1, a equality is also performed on predicates but the back propagation learning operation goes through the equality neurons. In this last case, the equality activation functions are to be carefully chosen.

In the case of simple evaluation, a good equality activation function is the Gaussian. But since the back propagation does not need to go through those neurons even a non continuous function like a combination of steps function will be satisfactory.

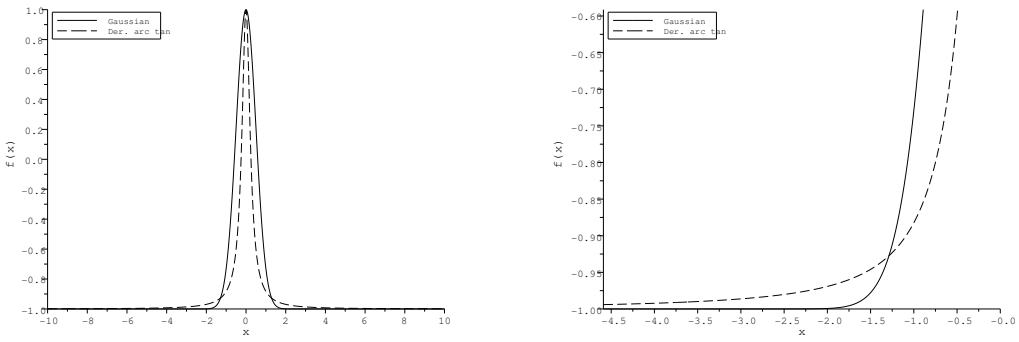
$$v = 2.e^{-\alpha.\|i\|^2} - 1$$

Where v is the output value of the neuron and i the sum of the inputs.

However, in the case of technique 2, because the Gaussian becomes null quite rapidly according to the precision of the computer, the back propagation algorithm cannot handle the learning task. For example consider two terms with a *distance* of 10 units. The equality test is almost -1 (which is what we expected) but the derivative is also almost null ($e^{-10^2} \approx 4.10^{-44} \approx 0$ for computer).

Base on this observation, the derivative of arc tangent has been considered. This function has the same "shape" as a Gaussian, but the derivative is more important, and therefore is not approximated to zero by computers. In this case, the back propagation algorithm can handle the learning task with this activation function.

The figure 3.5 presents a comparison of the Gaussian (with $\alpha = 2$) and the Derivative arc tangent (with $\alpha = 4$) equality function.



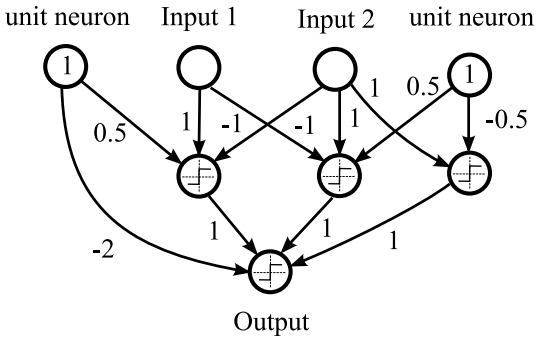
(a) Activation function (b) Zoom

Figure 3.5: Equality activation functions

The *extended equality* is an extension of the notion of equality. In the previously presented term convention, the value 0 represents a non existing term, and the following described techniques needs to consider two non existing terms unequal. In order to deal with this extension, an *extended equality* neuron will refer to a group of neuron that simulate equality as neurons with Gaussian or diff arc tangent activation function, but that

will not fire if the inputs have both the value 0.

The behavior of such neurons can be replicated with several simpler neurons. The figure 3.6 presents a group of neurons that achieve together an extended equality test and several runs of this ANN.



(a) The ANN

Input 1	Input 2	Output
0	0	-1
0	1	-1
1	0	-1
1	1	1
1	2	-1
2	1	-1
2	2	1

(b) The runs

Figure 3.6: Extended equality

3.4 Formula rewriting

In order to be handled by the system, the *initial background knowledge* rules have to be rewritten into an equivalent normal form. This rewriting from is also very close of the artificial neural network architecture, and it is indispensable to understand the underlying process.

This rewriting convert a rule A into an equivalent rule B, with the body of B expressed as a set of atoms with only free independent variables as argument, and a set of test (generalization of equality) over these free variable.

Given the general rule:

$$P_1(a_{1,1}, ..., a_{1,n1}) \wedge ... \wedge P_m(a_{m,1}, ..., a_{m,n_m}) \Rightarrow Q(b_1, ..., b_q)$$

The rewritten version of this rule (called the normal form) is :

$$P_1(c_1, ..., c_{n_1}) \wedge ... \wedge P_m(c_{n_1+...+n_{m-1}+1}, ..., c_{n_1+...+n_m}) \wedge (c_{d_1} = c_{d_2}) \wedge ... \wedge (c_{d_r} = c_{d_{r+1}}) \Rightarrow Q(c_{n_1+...+n_{m+1}}, ..., c_{n_1+...+n_{m+n}})$$

with all the C_i different. The body of a rewritten rule contains $c_{d_i} = c_{d_{i+1}}$ with $c_{d_i} = a_{j,k}$ and $c_{d_{i+1}} = a_{l,m}$ if and only if $a_{j,k} = a_{l,m}$

The underlying idea of the rewriting is to encode every term relation with an equality.

Example 3.8. Here are two simple examples of rule rewriting.

$$P(X,Y) \wedge Q(X) \Rightarrow R(Y)$$

becomes

$$P(X1,X2) \wedge Q(X3) \wedge (X1 = X3) \wedge (X4 = X2) \Rightarrow R(X4)$$

and

$$P(X,Y) \wedge P(Y,Z) \Rightarrow P(X,Z)$$

becomes

$$P(X1,X2) \wedge P(X3,X4) \wedge (X2 = X3) \wedge (X1 = X5) \wedge (X4 = X6) \Rightarrow R(X5,X6)$$

If the equations contain functions , the rewriting is extended with inversion of function:

If $X = f(a_1,...,a_m)$ is an instance of the function f, $f_n^{-1}(X)$ is the n^{th} argument of X i.e. $f_n^{-1}(X) = a_n$. If $n > m$ or if X is not a instance of the function f, $f_n^{-1}(X) = \emptyset$.

Example 3.9. $P(X, f(X)) \wedge Q(g(X)) \Rightarrow R(X)$ becomes $P(X1,X2) \wedge Q(X3) \wedge X1 = f_1^{-1}(X2) \wedge X1 = g_1^{-1}(X3) \wedge X4 = X1 \Rightarrow R(X4)$

$P(f(X,X)) \Rightarrow R(X)$ becomes $P(X1) \wedge X2 = f_1^{-1}(X1) \wedge X3 = f_1^{-2}(X1) \wedge X2 = X3 \wedge X2 = X4 \Rightarrow R(X4)$

This rewriting gives a direct way to encode a predicate logic rule into an ANN with the equality and extractor neurons. The exact convention is given in the next section.

3.5 Multi-dimensional neurons

In the networks presented several non common multi-dimensional activations functions are used. In a strict mathematical may, they can always be emulated by several ANNs with a common with activation function.

The multi-dimensional neurons (neurons carrying a multi-dimensional value) are used to prevent the network converging into undesirable states. For example, suppose $index(a) = 1$, $index(b) = 2$ and $index(c) = 3$. If it only uses single dimension neurons, since $index(c) = index(a) + index(b)$, the network may try to produce the term c by "adding" a and b. In order to avoid this kind of solution multi dimensional neurons are used in the following way.

Definition 3.4. *Set multi-dimension* (or *multidim*) is an activation function $\mathbb{R} \to V$, with V a a vectorial space.

Let $\{v_i\}_{i \in \mathbb{N}}$ be a free and generative family of V.

$$multidim(X) = v_{\lfloor X \rfloor}$$

Definition 3.5. *Set single dimension activation function* (or *invmultidim*) is an activation function $V \to \mathbb{R}$ that extracts the index of the greatest dimension of the input.

$$invmultidim(X) = i \text{ with } \forall j \; v_i.X \geq v_j.X$$

The following property holds $invmultidim(multidim(X)) = \lfloor X \rfloor$.

Example 3.10. Suppose $index(a) = 1$ and $index(b) = 2$.

$$multidim(index(a)) = v_{\lfloor index(a) \rfloor} = v_1 \tag{3.1}$$
$$multidim(index(b)) = v_{\lfloor index(b) \rfloor} = v_2 \tag{3.2}$$

multidim and *invmultidim* functions can be simulated with common neurons. However, for computational efficiency, it is extremely interesting to have special neurons for those operations.

The way to represent the multi-dimensional vectorial values is also an important point. For a given multi-dimensional vectorial neuron, the number of non null dimension is 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 $v_{10^{10}}$. It is therefore very important to not represent those multi dimensional spaces as an array indexed by the dimension number, but to use a mapping (like a hash table) between the index of the dimension and its component.

Definition 3.6. *Multi-dimensional sum activation function* (or *multidimSum*) is an activation function $\mathbb{R}^* \times V^* \to V$ that computes the sum of several multi-dimensional values and multiplicate it according to the following formula.

Suppose n a neuron with the *multidimSum* activation function.

$$V(n) = \sum_{\substack{(i \in InputConnectionNodes(n)) \\ \text{and} \\ (i \text{ is a multi dimensional neuron})}} (w_{i \to n}.V(i)).\sum_{\substack{(i \in InputConnectionNodes(n)) \\ \text{and} \\ (i \text{ is a single dimensional neuron})}} (w_{i \to n}.V(i))$$

Chapter 4

Technique 1

This technique is the first induction technique. Like the other techniques, the inputs of this technique a set of training example and the initial language bias. The output is a trained ANN that reacts correctly on the training example and generalize its knowledge in the case of new examples.

The global use is the following one:

1. Choose restrictions on the rules the system will learn (called setting the language *bias*)
2. Construct a special artificial neural network using the language bias
3. Encode in the network the initial background knowledge (which can be empty)
4. Train the ANN on a set of examples encoded using the input convention defined page.
5. Evaluate the learned rules on set of tests
   If the evaluation or the training fails, start again using weaker language bias.
6. Extract the learned rules

A training example is a set of input atoms associated with a set of output atoms.
The training of ANNs produced with this technique is done in the following way:

For every epoch

For every training example = $(Input, Expected\ output)$

1. Reset the values of the network
2. Load the Inputs into the network
3. Load the expected outputs into the network
4. For $i = 1$ to N, where N is a parameter of the algorithm
   (a) Make a run with the network
   (b) Apply the back propagation algorithm

The learning process is based on the following feature: The ANN contains *error neurons* that should have a value as close as possible to 0. Therefore, the back propagation algorithm try to fix their value to 0. The value of the error neurons is computed base on the difference between the literals generated and the actually expected literals.

The ANN is divided into three parts:

1. *The input part* that load and store the input literals.
2. *The output part* that load and store the expected output literals.
3. *The rule part* that apply the rules encoded in the ANN to literals contained in the input part, and compare the result with the literals contained in the output part.

The input and expected output data are loaded and stored in the ANN a the beginning of the training of every training example. The storage operation is done with several *loops* of neurons. The notions of "loop" is explained in the next section.

The figure 4.1 is a graphical representation of the network that is created and trained in the general case. The label "= 0" are for the *error neurons* for which the value should be 0.

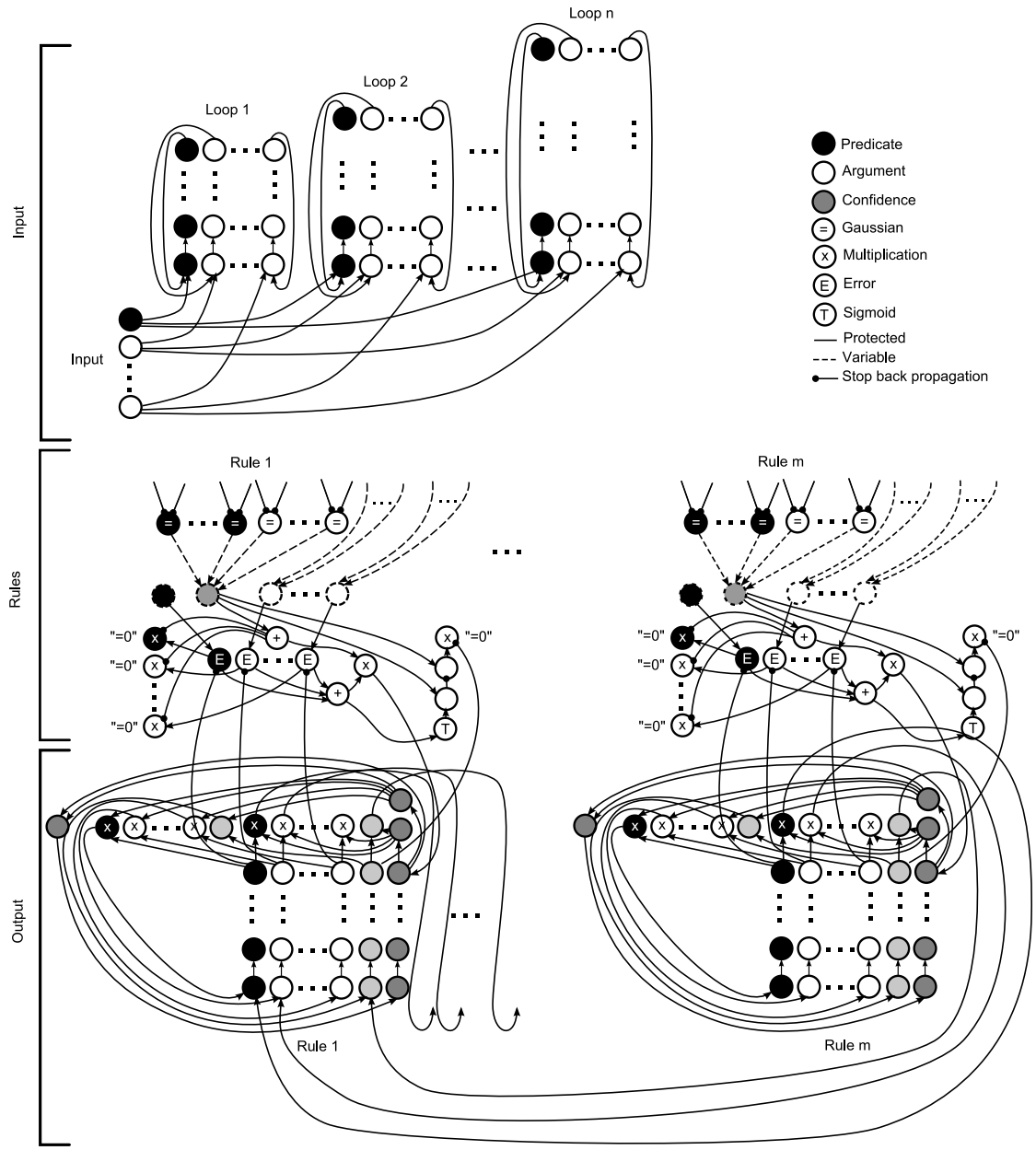


Figure 4.1: graphical representation of the meta network

Since this technique is just an introduction to the next techniques, the formal description of the network is not given here. An informal presentation of the expected behavior is given in the next section. The last section discusses and analyze the problem of this technique.

4.1 The *loops*

A loop is an architectural definition of a part of the artificial neural network. The goal of a loop is to store literals in the network, and to present them in a cyclic way to an another part of the network.

A loop is constituted of a list $\{S_i\}_{i\in[1,n]}$ of *stratums*, and a loading stratum S_0. At a time t, every stratum contains a literal or is empty. The activation function of all the neuron of a loop is the *addition* function. At a time $t+1$, for $i > 1$, the stratum S_{i+1} will contains the literal that was contained at the time t in the stratum S_i, and the stratum S_1 will contains the literal that was contained at the time t in the stratum S_n, or the literal contained in the stratum S_0. A loop of size n cannot contains more that n literals. The loading of the literal in the loop is done by presenting a literal to the stratum S_0, followed by a run of the network.

A stratum is composed of a neuron that stores the index of the predicate of the stored literal, a neuron that stores the "sign" of the stored literal, and m neurons that store the $p \leq m$ argument of the stored literal.

The next figure shows the comportment of a simple loop at three different times ($t = 1$, $t = 2$ and $t = 3$). Two runs are presented here.

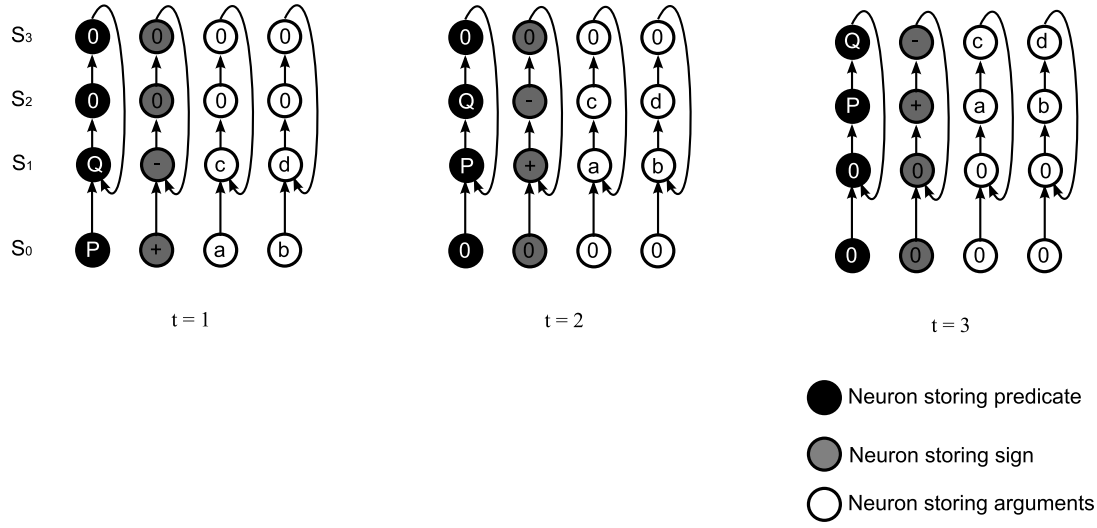


Figure 4.2: Example of loop

Once a loop is loaded with literals, the neurons of the stratum S_0 are set to the value 0. At every new run of the network, a new literal is presented to the bottom stratum S_1.

If the run of the network is continued without presenting any we literal to the stratum S_0, the two literals $P(a, b)$, $\neg Q(c, d)$ and the empty literal $0(0, 0)$ will turn in the network. The literal $P(a, b)$ will be presented in S_1 at the times $t \in \{2, 5, 8, 11, ...\} = \{2 + 3 \cdot k | k \in \mathbb{N}\}$. The literal $\neg Q(c, d)$ will be presented in S_1 at the times $t \in \{1, 4, 7, 10, ...\} = \{1 + 3 \cdot k | k \in \mathbb{N}\}$.

4.2 Input and output convention of the network

The input layer of the network contains three types of neurons: one *predicate neuron*, one *sign neuron* and *maxArity argument neurons*. The next algorithm describe how to "load" a set of input literal I, and a set of expected output literal O into the network.

1. Give for every predicate a unique index (natural)
2. For each element $i \in I$
   (a) If i is a positive literal, set the value of the *sign neuron* to 1

(b) If i is a negative literal, set the value of the *sign neuron* to -1

(c) Set the value of the *predicate neuron* to the index of the predicate of i

(d) For $j = 1$ to n, with n the arity of the predicate of i

i. Set the value of the j^{th} *argument neuron* to $encode(A)$, where A is the j^{th} argument of i.

(e) Run a step of the network

3. Set $i = 1$

4. For each element $o \in O$

(a) Set s to be the i^{th} sign neuron of the first output loop

(b) Set p to be the i^{th} predicate neuron of the first output loop

(c) Set $\{a_j\}$ to be the argument neurons of the i^{th} layer of the first output loop

(d) If o is a positive literal, set the value of the *sign neuron* s to 1

(e) If o is a negative literal, set the value of the *sign neuron* s to -1

(f) Set the value of the *predicate neuron* p to the index of the predicate of i

(g) For $j = 1$ to n, with n the arity of the predicate of o

i. Set the value of the *argument neuron* a_j to $encode(A)$, where A is the j^{th} argument of o.

(h) Set $i = i + 1$

4.3 Informal presentation of technique 1

The technique works in the following way:

Input literals are loaded into the different input loops. Since these loops have different sizes, the literals at the bottom of the loops are always different. If the sizes of the different loops are correctly chosen, all the permutation of literals will be presented in the bottom of the loops through a finite number of runs of the ANN.

The rule layers of the network compare the literal of the expected output loops, and the literals computed from the ones presented at the bottom of the input loops. The difference between the expected literal and the constructed literal induce an error that drives the back-propagation algorithm. This error is also used in the output layers

The outputs layer distributes the expected output literal into the different output loops. Every loop corresponds to a different rule that can be learned. The more an expected output literal "fits" to a rule, the more it will stay in the loop of the rule, and the more the back propagation will shape the rule to produce the good output literal. This operation is done thanks to a *score* counter associated to every expected output literal. The worst a literal fits to a rule, the fastest its score increase. When the score of a literal reaches a given critical point, it is set to 0 and the literal is moves to a different loop i.e. to a different rule. Statistically, the literals will be sorted according to the rule that explains them the best.

4.4 Discussion about the limitation of this technique

This technique has a big problem. Most of the time, the network does not converge and none of the rules are ever explained. This behaviour comes from the fact that ANNs are not magical back box mathematical objects. The next techniques are focusing on a deeper understanding of the ANN and a better presentation of the data in order to help the ANN to converge.

Chapter 5

Technique 2

This technique is an improvement of the first technique. The loops have been deleted and replaced by a single layer. All possible combinations of input are presented to this layer.

In order to help the convergence, the atoms with the same predicate are grouped and presented to the same inputs (in the technique 1, all the atoms were mixed).

5.1 Construction and use of the network

In this section is described the construction of the artificial neural network that is the core of this induction technique.

At the end of this section, the figure 5.2 gives a graphical representation of a simple induction ANN.

There is one block of input neurons in the network for every possible input atoms, an activation level neuron, a sign neuron and a number of arguments neurons corresponding to the arity of the predicate. There is on block of output neurons for every possible output atoms, an activation level neuron, a sign neuron and a number of arguments neurons corresponding to the arity of the predicate.

To build the network, we define some restriction on the rules that can be learned:

> $prePred$ = the maximum number of atoms with the same predicate allowed in the body of the rule
>
> The number of layer and the number of neuron on every layer, for the *internal activation network*. For most of the cases, a single neuron (one layer with one neuron) if enough. The *internal activation network* is a common sub artificial neural network.

If those restrictions are too strict, the training will fail. However, in this case, a simple solution is to start the training again on a less strict set of restrictions.

Regarding the examples, the set of input predicate $\{P_i\}_{i\in\mathbb{N}}$ and the set of output predicate $\{Q_i\}_{i\in\mathbb{N}}$ is extracted.

Example 5.1. With the following training example, the input predicates are P and Q, and the only output predicate is R.

$$P(a) \wedge Q(a) \Rightarrow R(a)$$
$$P(b) \wedge Q(b) \Rightarrow R(b)$$
$$P(c) \wedge Q(c) \Rightarrow R(c)$$

Remark 5.1. The network is based on the predicates known from the training examples. If during the evalutation, a new predicate appears, it will simply be ignored.

Based on those data, the following artificial network is constructed:

Algorithm 12 Building of the Induction ANN

1. For every input predicate P_i and for every $j \in [1, prePred]$
   (a) Create the input unit $P_{i,j}\_Activation$
   (b) Create the input unit $P_{i,j}\_Sign$
   (c) For $k \in [1, N]$ with N the arity of the predicate P_i
      i. Create the input unit $P_{i,j}\_Arg_k$
      ii. Create the unit $P_{i,j}\_Arg_k\_M$ with the activation function $multidim$
      iii. Connect $P_{i,j}\_Arg_k$ to $P_{i,j}\_Arg_k\_M$ with a weight connection of 1
2. For every output predicate Q_i
   (a) Create the output unit $Q_i\_Activation$
   (b) Create the output unit Q_i_Sign
   (c) For $k \in [1, N]$ with N the arity of the predicate Q_i
      i. Create the output unit $Q_i\_Arg_k$ the activation function sum
      ii. Create the output unit $Q_i\_Arg_k\_M$ the activation function $invmultidim$
      iii. Connect $Q_i\_Arg_k\_M$ to $Q_i\_Arg_k$ with a weight connection of 1
      iv. Connect all the $P_{i,j}\_Arg_k$ neuron to the unit $Q_i\_Arg_k\_M$ with a random small positive weight.
3. For all possible pair of different neurons $P_{i,j}\_Arg_k$
   (a) Create a equality unit without bias
   (b) The activation function of this unit is a Gaussian or $\frac{d(arctan(x))}{dx}$ (see section 3.3).
   (c) Connect the two neurons of the pair to this unit with a protected link of weight 1 and -1.
4. Create the internal activation network (number of layers and number of unit per layer as defined in the configuration).
5. All the neuron of the internal activation network are a bi-sigmoid activation function.
6. Connect all the *equality* units to all the input nodes of the *internal activation network*.
7. Connect all the *sign* and *activation* units to all the input nodes of the *internal activation network*.
8. Connect all the output unit of the *internal activation network* to all the *sign* and *activation* unit of the output nodes.

The figure 5.1 presents a graphical general representation of the network for an only output predicate Q_i.

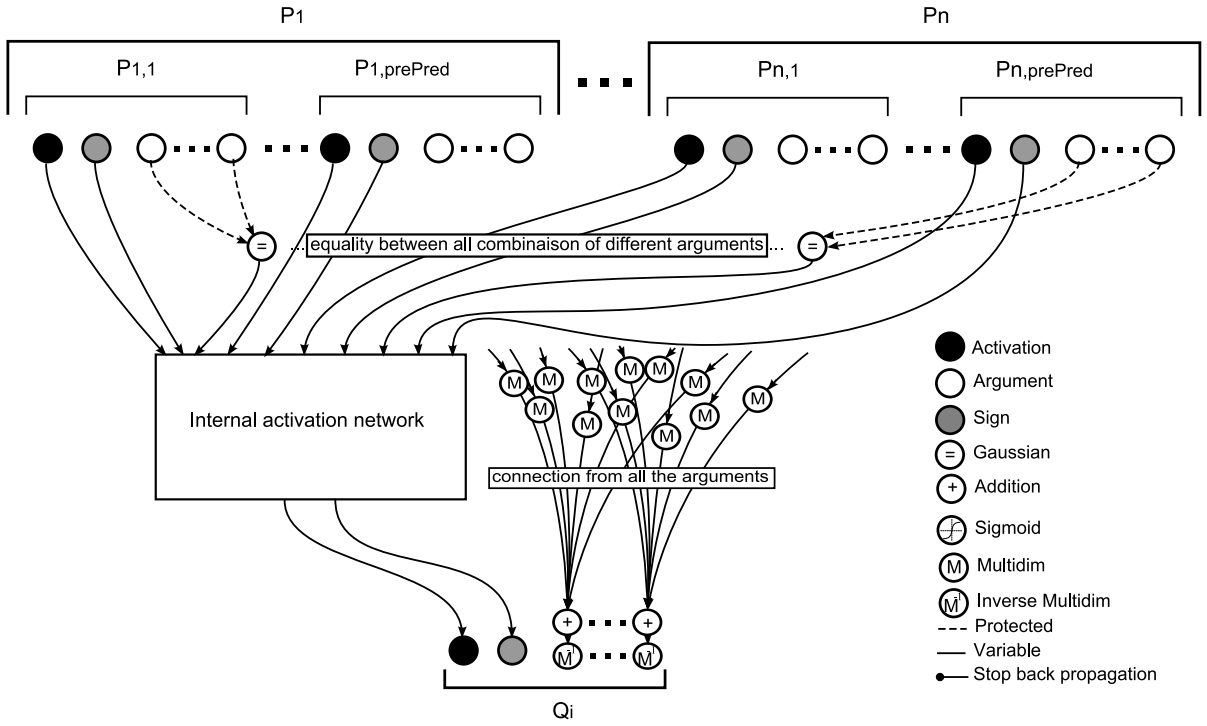


Figure 5.1: Second technique artificial neural network

In the case of initial background knowledge, the initial network has the same architecture with edges weights are different.

The following procedure presents the weight assigning to do to incorporate initial background knowledge defined as a rule in with previously defined rewriting convention i.e.

$$P_1(c_1, ..., C_{n_1}) \wedge ... \wedge P_m(c_{n_1+...+n_{m-1}+1}, ..., c_{n_1+...+n_m}) \wedge c_{d_1} = c_{d_2} \wedge ... \wedge c_{d_r} = c_{d_{r+1}} \\ \Rightarrow Q(c_{n_1+...+n_{m+1}}, ..., c_{n_1+...+n_{m+n}})$$

With $\{P_i\}_{i\in\mathbb{N}}$ the input predicates and $\{Q_i\}_{i\in\mathbb{N}}$ the outputs predicates.

With this algorithm, a single rule can be given for every output predicate as background knowledge. With the same approach of the next algorithm, an algorithm that is able to encode several back ground knowledge rules can be build. This algorithm is not presented here.

Algorithm 13 Including initial back ground knowledge in an Induction ANN

1. Set Pos an empty list of neuron
2. Set Neg an empty list of neuron
3. Set M an empty list of neuron
4. For all atoms P_i in the body of the rule
   (a) Found j the smallest j with $P_{i,j}\_Activation$ not in M
   (b) Add $P_{i,j}\_Activation$ to M
   (c) If the atom is a "negative" atom (P_i^*), add the neuron $P_{i,j}\_Sign$ to Neg
   (d) If the atom is a "positive" atom (P_i^*), add the neuron $P_{i,j}\_Sign$ to Pos
   (e) Add the neuron $P_{i,j}\_Activation$ to Pos
5. For all equality between X_l and X_m in the body of the rule
   (a) If the two part of the equality are defined in the input predicates
      (b) Add the equality neuron u with input connection from the neurons v and w, to Pos
      With v the neuron $P_{i_l,j_l}\_Arg_{k_l}$ associated with X_l and
      ii. w the neuron $P_{i_m,j_m}\_Arg_{k_m}$ associated with X_m
   (c) If the one of the part of the equality are defined in the output predicates
      (d) Set the weight of the edge between u and v to 1
      With u the neuron $P_{i_l,j_l}\_Arg_{k_l}$ associated with X_l and
      ii. v the neuron $Q_{i_m}\_Arg_{k_m}$ associated with X_m
   (e) If Q_i, the output atom is positive, set bias of the neuron Q_i_Sign to 1
   (f) If Q_i, the output atom is negative, set bias of the neuron Q_i_Sign to -1
   (g) Commentary: The target is now to make the Internal activation network to fire if an only if all the neurons in Pos have a value greater than 0.5, and all the neurons of Neg have a value lower than -0.5.
   (h) Compute $A = \{a_i\}$ the set of the first neuron of every hidden layers of the Internal activation network
   (i) With a_i the frst neuron of the i^{th} hidden layer
   (j) For all connection between neurons of A, set the weight to 1
   (k) Set the connection between the last neuron of A and the neuron $Q_i\_Activation$ to 1
   (l) For all neuron $n \in Pos$
      i. Set the connection between n and the first neuron of A to 1
   (m) For all neuron $n \in Neg$
      i. Set the connection between n and the first neuron of A to -1
   (n) Set the bias of the first neuron of A to $0.5 - |Pos| - |Neg|$

The figure 5.2 is graphical representation of a induction ANN containing the initial background knowledge $P(X,Y) \wedge Q(X) \Rightarrow R(Y)$.

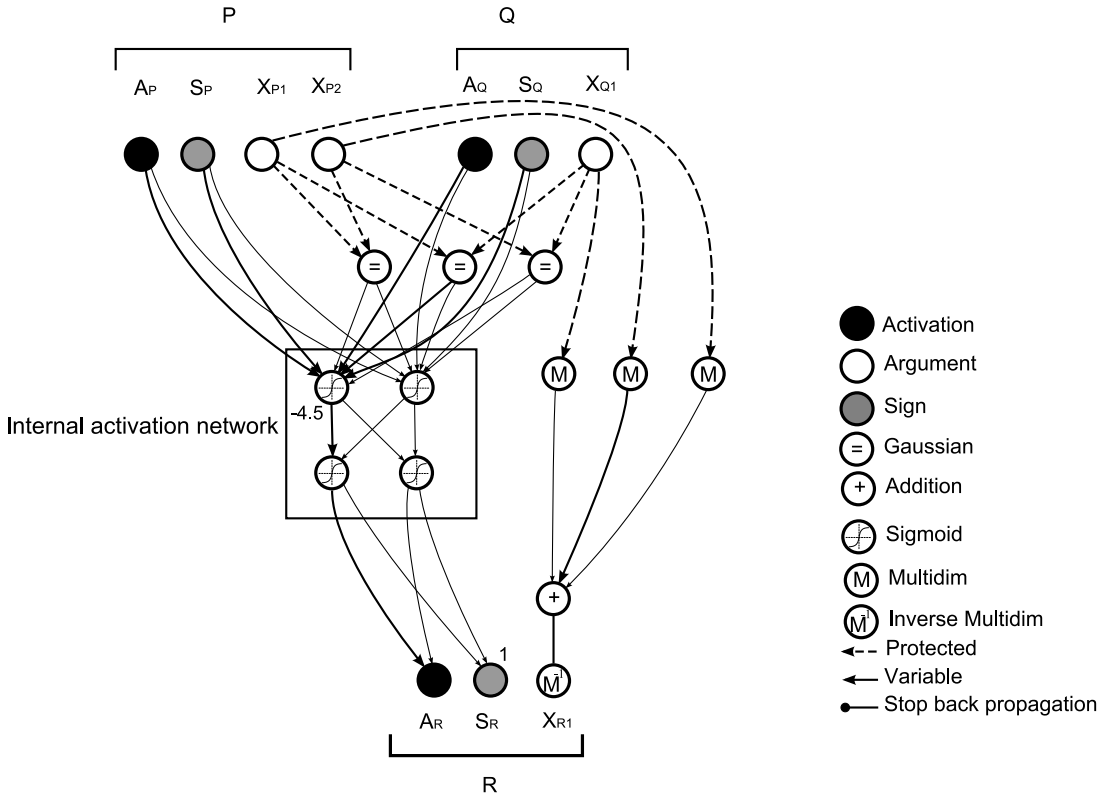


Figure 5.2: Example of induction ANN

The small edges have weight close to zero and the large edges have weight close to 1. The side number of each neuron is the bias (weight of connection from the unit neuron). If the bias is not given, it is close to zero.

To help the visualisation, the figure 5.3 is the same graph without the edges with close to zero weights, and without the hidden nodes without strong output connections.

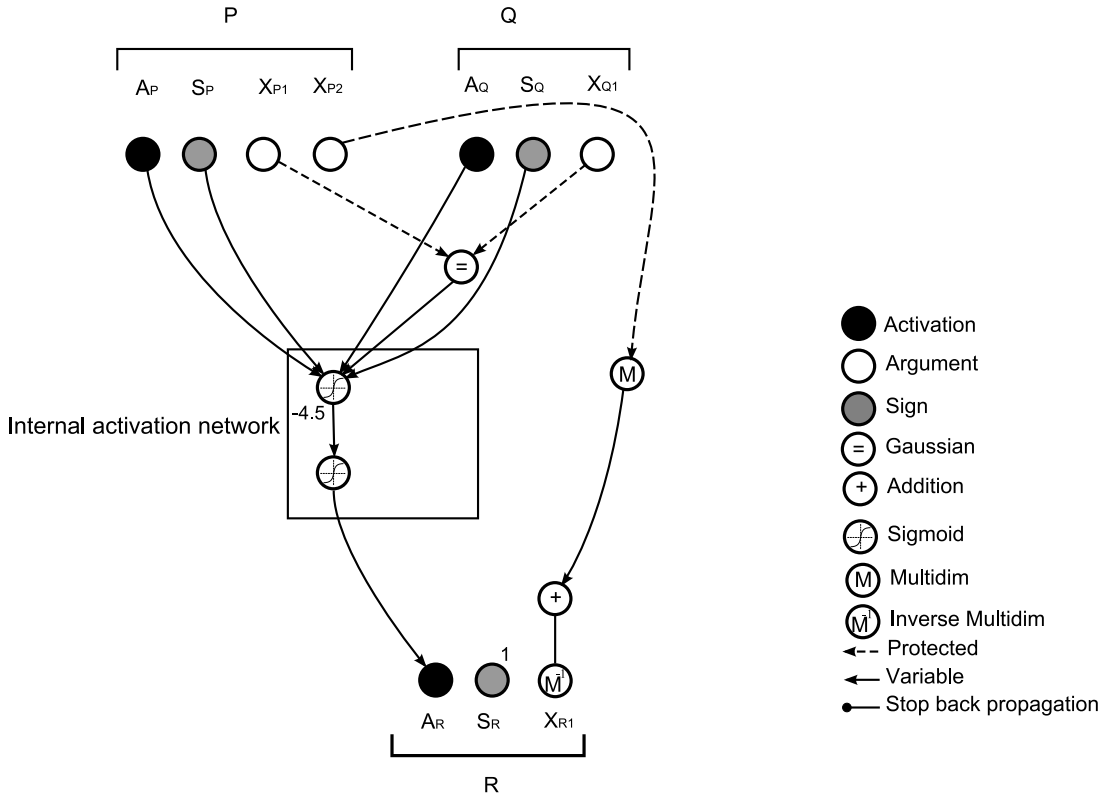


Figure 5.3: Example of induction ANN

The following lines present an example of running of this ANN. The exact input and output convention is given in the next section.
In this ANN, the sign neuron of the output predicate S_R has always a value of 1 i.e. positive.
The activation neuron of the output predicate S_R fire if and only if:

The input predicate P is present i.e. the input neuron A_P has a value of 1.

The input predicate P is positive i.e. the input neuron S_P has a value of 1.

The input predicate Q is present i.e. the input neuron A_Q has a value of 1.

The input predicate Q is positive i.e. the input neuron S_Q has a value of 1.

The first argument of P is equal to the first argument of Q i.e. $X_{P_1} = X_{Q_1}$.

In this case we read:
The sign neuron of the output predicate S_R that has always a value of 1 i.e. positive. The argument of the output predicate X_{R_1} is equal to the second argument of the input predicate P i.e. $X_{R_1} = X_{P_2}$

5.2 Training and evaluation

In the following lines is presented the convention on the values that have to be loaded in the network.

Suppose an atom $A = P(x_1, ..., x_n)$ associated with a set of neuron $\{N_i\}$ having the corresponding predicate P:

The sign neuron's value is assigned to 1 if A is a positive atom, and -1 if A is a negative atom.

The activation neuron's value is assigned to 1.

For every input argument neurons N_j, the value is assigned to $encode(X_j)$.

For every output argument neurons N_j, the expected value is $encode(X_j)$. multidimensional space.

Suppose a set of neuron $\{N_i\}$ having the corresponding predicate P. If $\{N_i\}$ have no atom assigned:

The sign neuron's value is assigned to 0.

The activation neuron's value is assigned to -1.

For every input or output argument neurons N_j, the value is assigned to 0. In the case of an output neuron, the backward propagation, the error is null.

For the training and the evaluation, all the possible combinations of association are trained and tested. In the case of the training, to avoid the *symmetrical problems*, a decreasing learning rate is used through all the combination of a given input data set. The current used solution is $\epsilon = \frac{\epsilon_{base}}{\sqrt{combination\_number}}$.

An interesting restriction can be to do not allow input atoms to be used more than once in a rule. This restriction is often good and helps a lot the system to converge.

Definition 5.1. The *symmetrical problems* appears when several atoms with the same predicates are presented in the same time to the network trough several symmetrical positions. The resulting inferred rule is a meaningless fusion of the expected rule with different but equivalent organization of the body.

For example, the rule

$$P(X_1, X_2) \wedge P(X_3, X_4) \wedge X_2 = X_3 \wedge X_5 = X_1 \wedge X_5 = X_4 \Rightarrow P(X_5, X_6)$$

is equivalent to the rule

$$P(X_1, X_2) \wedge P(X_3, X_4) \wedge X_1 = X_4 \wedge X_5 = X_3 \wedge X_5 = X_2 \Rightarrow P(X_5, X_6)$$

Therefore, if $P(a, b) \wedge P(b, c) \Rightarrow P(a, c)$ is presented to the system. Both of those equivalent rules will be reenforced, and the resulting rule can become for example $P(X_1, X_2) \wedge P(X_3, X_4) \wedge X_2 = X_3 \wedge X_5 = X_3 \wedge X_5 = X_4 \Rightarrow P(X_5, X_6)$.

A changing learning rate allows giving advantage to one of them and breaking the symmetrical problem.

5.3 Extensions

Several extension of the architecture can be done to increase the expression power of the learned rules. Those extensions increase the complexity of the network and can sometime, generate convergence problems. They are all independent.

5.3.1 Ground terms in the rules

The previously presented architecture does not allow rules with ground term in the body or in the head. This restriction is often very useful because it allows a better generalization. However, the following extension amits this restriction.

This extension is a special case of the extension *Functions in the body and in the head* presented in the next section.

5.3.2 Functions in the body and in the head

The previously presented architecture can't handle rules with functions in the body or in the head, even if they are allowed as input.

The following extension allows inferred rules through a depth analysis of the functions. For example, it allows to learn rules of the kind $P(X, f(X)) \wedge Q(g(X, f(X)) \Rightarrow R(f(f(X)))$.

This extension increases significatively the complexity of the network but it does not generate convergence problems.

A new parameter of the construction of the artificial neural network is added. $maxDepth$ is the maximum depth of terms in the rules. For example, if $maxDepth = 1$, terms like $f(g(f(g(a))))$ can be handled by the system but the learned rules will not contain terms with a depth that 1 (like $f(g(X))$ or $f(X, g(Y))$ for example).

Definition 5.2. If $s = encode(f(x_0, ..., x_n)) = E'(index(f), E''(index'(x_0)), ..., E''(index'(x_n)), 0)$ is the input of an extraction neuron E^{i+1}, the output value of this neuron is $o = encode(x_i) = E''(index'(x_i))$ is $i < n = D_1(i)$, and $o = 0$ otherwise ($o = D_1(D_2^{i+1}(s))$).

To the previously defined ANN, we add the following neurons.

Algorithm 14 Adding function in the head and body extension

1. Set L a list of all the arguments inputs
2. Set L'' a list of all the arguments inputs
3. Set F the list of all functions (ground term of arity greater that zero) of the training examples
4. If we want to allow ground term in the rule definition, set F the list of all functions and constants of the training examples
5. For $i \in [1, maxDepth]$
   (a) Set $L' = L$
   (b) Clear L
   (c) For $j \in L'$
      i. Create a unit u with an activation function E^0
      ii. Connect j to u
      iii. For $f \in F$
         A. Create a equality unit e without bias
         B. The activation function of this unit is a Gaussian or $\frac{d(arctan(x))}{dx}$ (see the equality section for more details).
         C. Connect u to e with a protected link of weight 1.
         D. set the bias of the unit e to $-index(f)$.
      iv. For $k \in [1, maxFunctionArguments]$
         A. Create a unit u with an activation function E^j
         B. Connect j to u
         C. Add u to L
         D. Add u to L''
6. For all pair (i, j) of different element of L''
   (a) If an equality does already exist between i and j, continue
   (b) Create a equality unit without bias
   (c) The activation function of this unit is a Gaussian or $\frac{d(arctan(x))}{dx}$ (see the equality section for more details).
   (d) Connect the two neurons of the pair (i, j) to this unit with a protected link of weight 1 and -1.

The example 5 shows a learning run base on this extension.

The figure 5.4 presents a running example of a sub part of this architecture for the atom $P(a, g(b))$.

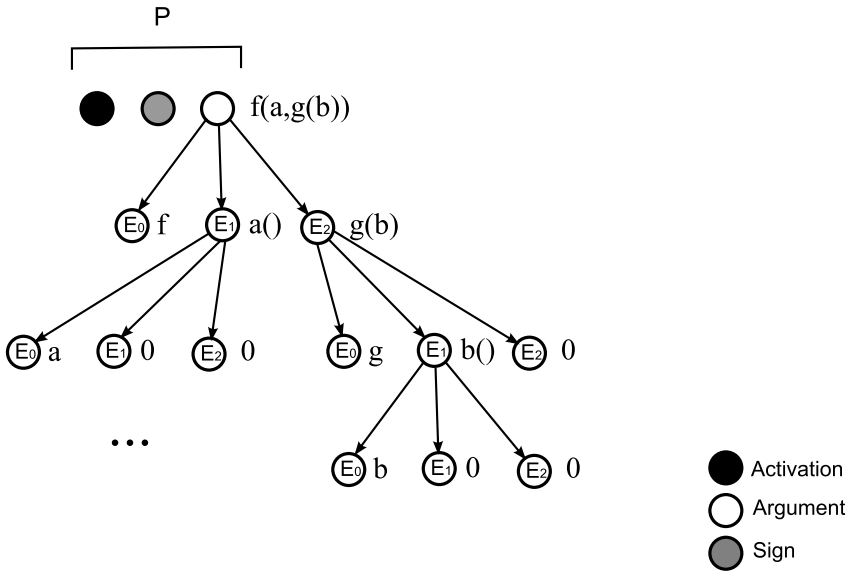


Figure 5.4: Function architecture

5.3.3 More expressive head argument pattern relations

The previously presented architecture can't handle rules with "complex" pattern relation over the argument of the head. For example the rules $P(X, X, Y, Z) \Rightarrow Q(X)$ and $P(X, Y, Z, Z) \Rightarrow Q(Z)$ can't both be leaned by the initial Inductive neural network. This is true whatever the initial set of restriction is.

The following extension deals with this restriction.

For this extension, the integer parameter *paternComplexity* should be fixed. If the *paternComplexity* is chosen to be 1, the expression power of the ANN will be the same as if this extension is not used. The larger this parameter is, the more complex the head argument pattern is.

Algorithm 15 Extending an Inductive ANN to handle complex head argument pattern relations

1. Delete all outputs connections of all neurons with the *multidimentional* activation function
2. For every output predicate Q_i, and every argument neurons $Q_i\_Arg_k\_M$ of Q_i
   (a) For $j \in [1, paternComplexity]$
      i. Create u a multidimensional neuron with the *sum* activation function
      ii. Connect u to $Q_i\_Arg_k\_M$
      iii. Connect all the neurons with the *multidimentional* activation function that are associated to Q_i, to u
      iv. Connect all the neurons of the last layer of the internal activation neuron to u.

In this extension, the internal activation network "select" which of the output argument should be chosen. Therefore, it is important to also increase the number of layers and the number of neurons on every layer of the internal activation network.

If this extension is used with any other extensions, this algorithm should be the last to be run.

Even through this extension does not increase a lot the complexity of the network, it does generate convergence problems.

5.3.4 Term typing

Considering typing of term can reduce the complexity of the network i.e. if two terms have different type, there is not meaning to test the equality between them. Therefore, by typing the terms, we can reduce the size of the network.

The term typing also permits to allow specialized condition in the network i.e. conditions that only have a meaning for some types of term. For example, we can add the integer $\leq$ comparison or the test of oddness for integers, or the extraction of the head and the tail for lists. In this way, the power of the induction can be increased.

The typing is defined as follow: For every argument of every predicate, a type is associated. Equality neurons between neurons that represent argument of different type are deleted. More of that, connections between neurons that represent argument of input predicates, and neuron that represent argument of output predicate with a different type are also deleted.

For every wanted typed comparison or function, neurons should be created for all possible combinations.

It is important to note that, by opposition to equality, general functions may not be symmetrical, and therefore, in this case, every rotation of argument should be tested.

For example, suppose the test $>$ on integers. The tests $a > b$ and $b > a$ should both be done. In this last case actually, because the equality is also present, the test $b > a$ is not really useful since it can be done with the test $a > b$ and the equality test $a = b$.

Another kind of term typing could also be done by associating to every function and constant a type. This way to deal with typing does not decrease the complexity of the network. It actually increases it a lot. But it is more expressive and allows for example to deal with typed recursive terms (For example, types for a list of lists).

5.3.5 Representation of lists

A simple way to represent lists is to use the Prolog convention. For example, the list $[1, 2, 3]$ will be represented with the term $list(1, list(2, list(3, \emptyset)))$. This encoding works but generated large number through the term encoding, because the head of the functions $list(\_, \_)$ should be represented for every element.

Choosing to give a special treatment for lists is very interesting for two reasons:

Firsttly, all the encoded values are smaller. Secondly, it allows having special functions and tests for lists like the “tail extraction” function or the computation of the size of the list.

A way to represent lists inspired from the term encoding is to represent every list $L = (e1, ..., e_n)$ by the integer $E'(L)$. As shows at the end of the term encoding section, page ?, the extraction of terms is extremely simple.

5.3.6 Rule dependence

In the current architecture, every rule is independent. But it can appear that some dependence between the rules can exist and help the learning process.

This extension allows dependence between the rules.

Example 5.2. It is easier to learn that:

$$A(X) \wedge B(X) \wedge C(X) \Rightarrow E(X) \tag{5.1}$$
$$D(X) \wedge E(X) \Rightarrow F(X) \tag{5.2}$$
$$\tag{5.3}$$

than to learn that:

$$A(X) \wedge B(X) \wedge C(X) \Rightarrow E(X) \tag{5.4}$$
$$A(X) \wedge B(X) \wedge C(X) \wedge E(X) \Rightarrow F(X) \tag{5.5}$$
$$\tag{5.6}$$

5.4 Examples of run

This section presents some example of running of this technique.

5.4.1 Example 1

This example shows the possibility to learn extremely simple rules.

(a) Training set	(b) Evaluation set
$P(a) \Rightarrow Q(a)$	$P(d) \Rightarrow Q(d)$
$P(b) \Rightarrow Q(b)$	$P(e) \Rightarrow Q(e)$
$P(c) \Rightarrow Q(c)$	$P(f) \Rightarrow Q(f)$

Table 5.1: Data set

The algorithm is run without any extension, with 2000 trainings, a learning rate of 0.01 and the number of predicate replication (*prePred*) of 2. The *internal activation network* is composed of two hidden layers with 3 neurons each. The same result is obtained with even smaller *internal activation network* (One hidden layer with one neuron).

5.4.2 Example 2

This example shows how produced arguments can be "selected" from the input terms.

(a) Training set	(b) Evaluation set
$P(a, a) \Rightarrow R(a)$	$P(m, n) \Rightarrow R(n)$
$P(a, b) \Rightarrow R(b)$	$P(m, o) \Rightarrow R(o)$
$P(a, c) \Rightarrow R(c)$	
$P(b, c) \Rightarrow R(c)$	
$P(b, a) \Rightarrow R(a)$	

Table 5.2: Data set

The algorithm is run without any extension, with 2000 trainings, a learning rate of 0.01 and the number of predicate replication (*prePred*) of 2. The *internal activation network* is composed of two hidden layers with 3 neurons each. The same result is obtained with even smaller *internal activation network* (One hidden layer with one neuron).

5.4.3 Example 3

This example shows how the relation (equality) between the different terms activates or inhibits the rule.

(a) Training set	(b) Evaluation set
$P(a,b) \wedge Q(a) \Rightarrow R(b)$	$P(m,n) \wedge Q(m) \Rightarrow R(n)$
$P(c,d) \wedge Q(c) \Rightarrow R(d)$	$P(m,n) \Rightarrow$
$P(e,f) \wedge Q(e) \Rightarrow R(f)$	$Q(m) \Rightarrow$
$P(a,b) \wedge Q(c) \Rightarrow$	$P(m,n) \wedge Q(o) \Rightarrow$
$P(c,d) \wedge Q(e) \Rightarrow$	
$P(e,f) \wedge Q(a) \Rightarrow$	
$P(a,b) \Rightarrow$	
$P(c,d) \Rightarrow$	
$P(e,f) \Rightarrow$	
$Q(a) \Rightarrow$	
$Q(c) \Rightarrow$	
$Q(e) \Rightarrow$	

Table 5.3: Data set

The algorithm is run without any extension, with 2000 trainings, a learning rate of 0.01 and the number of predicate replication (*prePred*) of 2. The *internal activation network* is composed of two hidden layers with 3 neurons each. The same result is obtained with even smaller *internal activation network* (One hidden layer with one neuron).

5.4.4 Example 4

This example shows how atoms with the same predicate can interact together thanks to a *variable learning rate*.

(a) Training set	(b) Evaluation set
$P(a,b) \wedge P(b,c) \Rightarrow R(a,c)$	$P(m,n) \wedge P(n,o) \Rightarrow R(m,o)$
$P(a,b) \wedge P(c,d) \Rightarrow$	$P(m,n) \wedge P(o,p) \Rightarrow$
$P(a,b) \Rightarrow$	$P(m,n) \Rightarrow$
$P(b,c) \Rightarrow$	$P(o,p) \Rightarrow$

Table 5.4: Data set

The algorithm is run without any extension, with 2000 trainings, a initial learning rate of 0.01 and the number of predicate replication (*prePred*) of 2. The *internal activation network* is composed of two hidden layers with 3 neurons each. The same result is obtained with even smaller *internal activation network* (One hidden layer with two neurons).

5.4.5 Example 5

This example shows how functions can be added in the body of a rule. In this case the rule is $P(X, f(X)) \wedge Q(g(X, f(X))) \Rightarrow R$.

(a) Training set	(b) Evaluation set
$P(a, f(a)) \wedge Q(g(a, f(a))) \Rightarrow R$	$P(d, f(d)) \wedge Q(g(d, f(d))) \Rightarrow R$
$P(a, f(b)) \wedge Q(g(a, f(a))) \Rightarrow$	$P(d, f(e)) \wedge Q(g(d, f(d))) \Rightarrow$
$P(a, f(a)) \wedge Q(c) \Rightarrow$	$P(d, f(d)) \wedge Q(f) \Rightarrow$
$P(a, f(a)) \wedge Q(g(c, f(a))) \Rightarrow$	$P(d, f(d)) \wedge Q(g(f, f(d))) \Rightarrow$
$P(a, f(a)) \wedge Q(g(a, f(d))) \Rightarrow$	$P(d, f(d)) \wedge Q(g(d, f(d))) \Rightarrow$
$P(a, f(a)) \Rightarrow$	$P(d, f(d)) \Rightarrow$
$Q(g(a, f(a))) \Rightarrow$	$Q(g(d, f(d))) \Rightarrow$

Table 5.5: Data set

The algorithm is run with the **Functions in the body and in the head** extension with a max depth of 2, with 2000 trainings, a learning rate of 0.01 and the number of predicate replication (*prePred*) of 1. The *internal activation network* is composed of two hidden layers with 3 neurons each.

5.4.6 Example 6

This example shows how functions can be used to deal with list. In this case the second letter has to be "d".

(a) Training set	(b) Evaluation set
$P(l(a, 0)) \Rightarrow$	$P(l(j, l(d, l(k, 0)))) \Rightarrow S$
$P(l(b, 0)) \Rightarrow$	$P(l(k, l(d, 0))) \Rightarrow S$
$P(l(c, 0)) \Rightarrow$	$P(l(d, l(k, 0))) \Rightarrow$
$P(l(a, l(d, 0))) \Rightarrow S$	$P(l(d, 0)) \Rightarrow$
$P(l(b, l(e, 0))) \Rightarrow$	
$P(l(c, l(f, 0))) \Rightarrow$	
$P(l(c, l(d, l(g, 0)))) \Rightarrow S$	
$P(l(b, l(e, l(h, 0)))) \Rightarrow$	
$P(l(a, l(f, l(i, 0)))) \Rightarrow$	

Table 5.6: Data set

5.4.7 Michalski's train problem

A N fold test have been done with the Michalski's train problem. At every run, one of the train is excluded from the training data, and put back for the evaluation i.e. "leave one out" cross-validation.

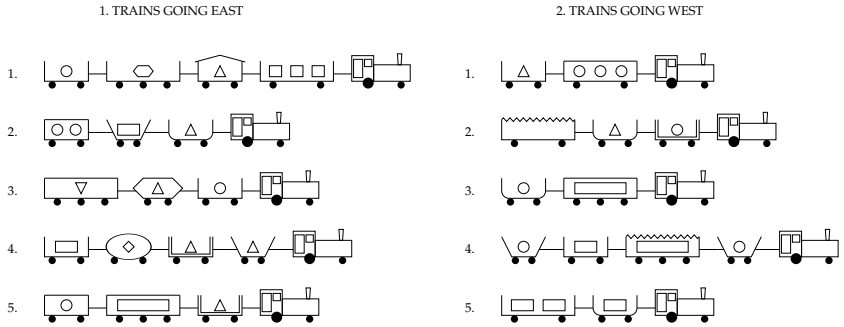


Figure 5.5: Michalski's train problem

Definition 5.3. The Michalski's train problem is a binary classification problem. The data set is composed of ten trains with different features (number of car, size of the cars, shape of the cars, object in the cars, etc.). Five of the train are going to the East, and the five other are going to the West. The problem is to found the relation between the features of the trains and theirs destination.

The table 5.7 presents a logic description of the first train.

Input		Direction (output)
Short(car2)	Closed(car2)	East
Long(car1)	Long(car3)	
Short(car4)	Open(car1)	
Infront(car1,car2)	Infront(car2,car3)	
Infront(car3,car4)	Open(car3)	
Open(car4)	Shape(car1,rectangle)	
Shape(car2,rectangle)	Shape(car3,rectangle)	
Shape(car4,rectangle)	Load(car1,rectangle,3)	
Load(car2,triangle,1)	Load(car3,hexagon,1)	
Load(car4,circle,1)	Wheels(car1,2)	
Wheels(car2,2)	Wheels(car3,3)	
Wheels(car4,2)		

Table 5.7: part of the Michalski's train problem Data set

One of the main features of the Michalski's train problem is the large number of irrelevant data. Because of the initial random values of the weighs of the network, all the simulations does not always give the same result. But most of the time (the N-Fold test has been run several time), the training examples are correctly assimilated (the evaluation on the training example give 100% of good results), but the new example seems to be almost randomly classified.

5.5 Discussion

This technique shows good results for simple instance of problem without irrelevant data (of very few). However, if is generally impossible to know exactly was data is important and what data is irrelevant. This point is therefore a serious issue.

The second important point is bonded with the notion of replication: If we allow the system to infer rule with several instance of atom with the same predicate, because of the way the different combinations are handled, the system reacts in the same way that it does when it meets irreverent data.

Chapter 6

Technique 3

The technique 2 shows some important deficiencies. In order to tackle them, several deep changes have been done, and the technique 3 has been created.

As an example, in the technique 2, if the architecture was defined to only learn a rule with one atom in the body, the training example $P(a,a) \wedge P(a,b) \wedge P(b,a) \Rightarrow R$ has to be divided into three different sub examples ($P(a,a) \Rightarrow R$, $P(a,b) \Rightarrow R$ and $P(b,a) \Rightarrow R$) to be used. Since the rule can only have one atom in the body, the work of the ANN is to choose which of the atom (if any) of the training example, is important. If a lot of irrelevant atoms are presented, the ANN may not learn correctly the rule.

The first section is an informal presentation of the technique 3 and its new features.Section 6.2 describes formally this technique. Section gives 6.3 a complete but simple example of instance of a use of this technique. Running example of this technique on different instances of problem is given in section 6.6.

6.1 Informal presentation

The technique 3 proposes an alternative way to load data in the network: Since an entire example cannot generally be loaded in the network, it has to be divided in some kind of way. *Memory neurons* are introduced to keep 'in mind' the relevant information of the different pieces of the input.

In the previous architectures, the back propagation algorithm has to be run several times on every example (once for every piece of example's input). Thanks to the memory neurons, the technique 3 needs only one run of back propagation for every example. This feature improve significatively the speed of the training.

This technique is an algorithm that uses one or several artificial neural network i.e. If a network is not 'good', it is destructed and a new one is constructed with different parameters i.e. allow more expressive rules to be learned.

The training of ANNs produced with this technique is done in the following way:

1. Do

   For every epoch
   (a) For every example
      i. Reset all memory neurons
      ii. For all the different distributions of the input of the example in the input layer of the network
         A. Load the configuration
         B. Make a run with the network
      iii. Apply the back propagation algorithm
2. If the learning is correct, Stop
3. Increase the expressive power of the rules to learn
4. While True

In comparison with the other techniques, the size of the network is increased. The following list presents the different layers of the produced networks. The layers marked with a star (*) are new layers specific to this architecture. The layer without star were already present with the technique 2. The exact definition of those layers and the explanation of their role is given in the formal definition of the technique.

1. Activation input and arguments layer

2. Extraction of the terms layer
3. *Application of function layer
4. Test layer (equality, inequality, $<$ for integers, etc.)
5. *Condition pattern layer
6. *Memory layer
7. *Output arguments memory layer
8. *Output arguments Extraction of the terms layer
9. *Output Application of function layer
10. Output arguments composition layer
11. Multi dimensional conversion layer
12. Activation ANN layer
13. Output arguments selection layer
14. Output activation and argument layer

Each of the layers of the network has a very specific task.

Example 6.1. Suppose the training example to be

$$\begin{aligned} P(f(a), a) &\Rightarrow R \\ P(f(a), b) &\Rightarrow \\ P(a, b) &\Rightarrow \\ P(b, a) &\Rightarrow \\ &\Rightarrow \end{aligned}$$

The following example presents informally how the system can learn the simple rule $P(f(X), X) \Rightarrow R$. Because this rule is extremely simple (no ground terms, no argument for the output predicate, a single atom in the body of the rule, etc.) most of the layer of the network are useless. Therefore, they are not considered in this example. However, the "fundamental" idea of the technique is still here.

Based on the rewriting convention every body of a rule can be equivalently rewrite as a set of atoms with only free independent variables as argument, and a set of test (generalization of equality) over these free variable. The rewriting version of the rule $P(f(X), X) \Rightarrow R$ is $P(A_1, A_2) \wedge (A_1 \text{ is a function } f) \wedge (f_1^{-1}(A_1) = A_2) \Rightarrow R$.

The network initially built for this learning has three input neurons : P is encoding the presence of an atom with the predicate P. A_1 is encoding the value of the term in the first argument of P (If there is a atom with the predicate P). A_2 is encoding the value of the term in the second argument of P (If there is a atom with the predicate P).

The *extraction of term layer* computes the decomposition of the terms presented in A_1 and A_2. For example if A_1 contains the term $f(a)$, the *extraction of term layer* compute the term a. But if A_1 contains a constant like a, the *extraction of term layer* does not compute anything from A_1.

The *test layer* computes the test of equality between the terms in A_1, A_2, and the terms computed in the *extraction of term layer* i.e $A_1 \stackrel{?}{=} A_2$, $(A_1 \stackrel{?}{=} f(X)) \wedge (X \stackrel{?}{=} A_2)$, $(A_2 \stackrel{?}{=} f(X)) \wedge (X \stackrel{?}{=} A_1)$, etc.

The *condition pattern* layer is not important in this example.

The *memory* layer remembers for every quality test T, if T has been activated at least once. For the rule to learn in this example, the important equality is $(A_1 \stackrel{?}{=} f(X)) \wedge (X \stackrel{?}{=} A_2)$. Let's call e the memory neuron that is linked to this test.

The *activation* layer is the part of the network that actually learn the rule. For this rule, this layer only needs to have one neuron l. All the neuron of the memory layer will be connected to this neuron, especially e. The neuron l is connected to a single neuron r of the output activation layer. This neuron represents the activation of the output predicate R i.e. if R is generated or not, by the system.

In the training, the neuron r needs to be activated if and only if the neuron e is activated i.e. the atom R is generated iff the test linked to e succeed. During the training, the back propagation algorithm "sees" this relation and adjust the weight of the connections between e, l and r in order to make r fires iff e fires.

The output of the technique is a network that produce the atom R iff the test $(A_1 \stackrel{?}{=} f(X)) \wedge (X \stackrel{?}{=} A_2)$ is verified i.e. if there is an atom with the predicate P, and if the decomposition of the term of the first argument of P is equal to the second argument of this atom i.e. $P(f(X), X) \Rightarrow R$.

The figure 6.1 presents an graphical representation of the important neurons for this rule.

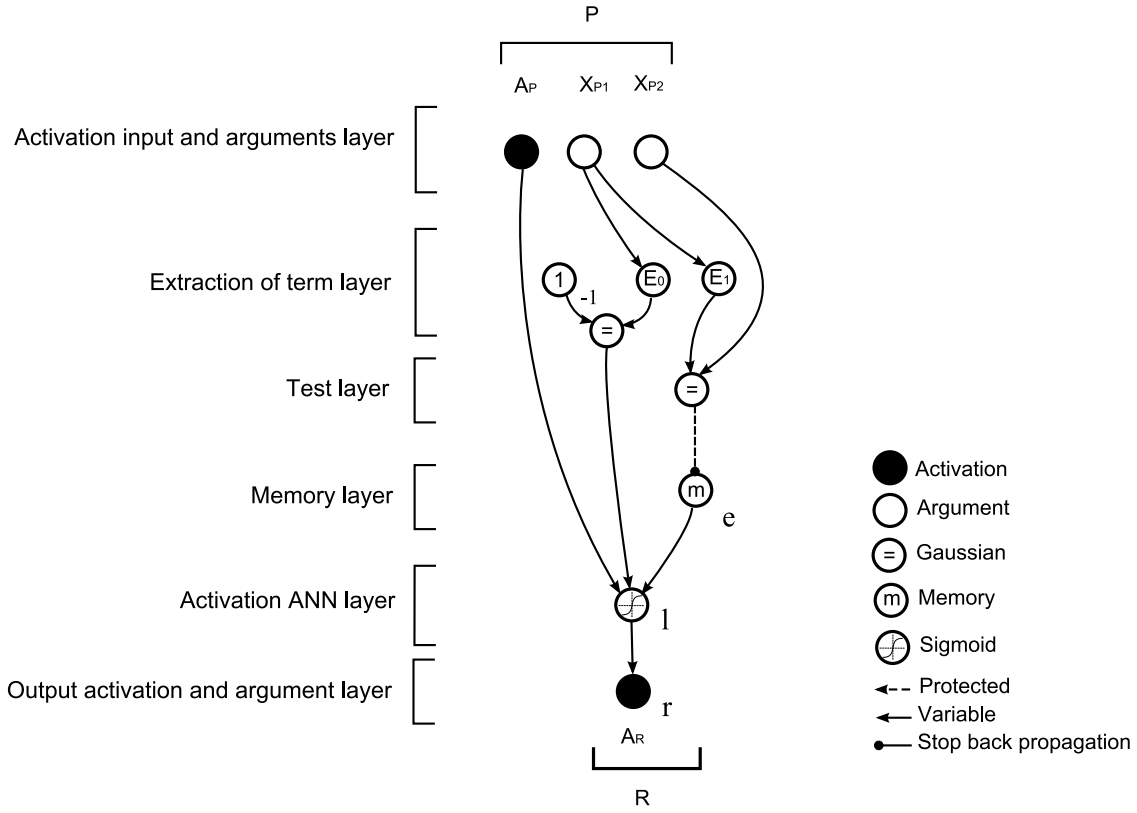


Figure 6.1: The important neurons

This example shows the basic idea of association between test, and generation of atom. For more complex rules, more complex patterns are used. In the case of output predicates with arguments, the back propagation is also connecting the good input term, or the good term composed from the input terms, to the corresponding output term.

6.2 Formal description

Definition 6.1. An *oracle neuron* is a neuron with a complex activation function used as an oracle. The back propagation should never go through them.

Definition 6.2. A *memory neuron* is a neuron that "remembers" if it was once activated. A memory neuron can be constructed with a simple recurrent neuron with the output connected to one of the input. The weight of this recurrent connection depends on the activation function of the neuron. The figure 6.2 represents a memory neuron with a step activation function.

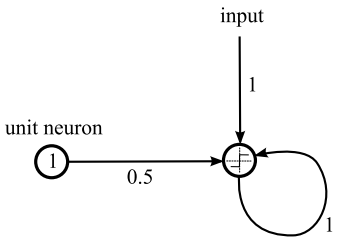
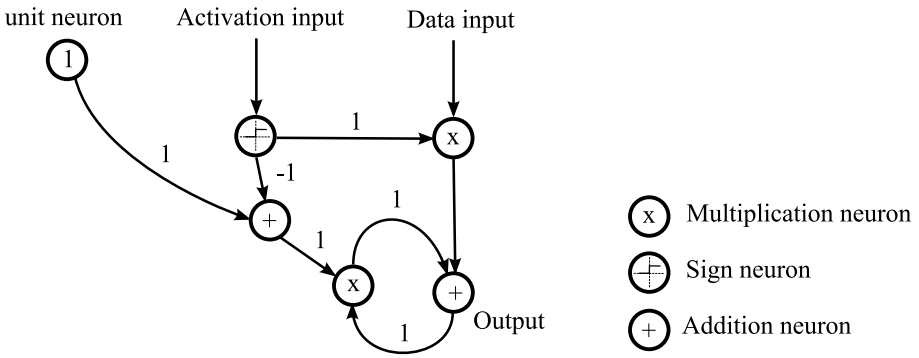


Figure 6.2: A memory neuron

Definition 6.3. An *extended memory neuron* is a neuron that 'remember' the value ($\mathbb{R}$) that was on its **data input** the last time its **activation input** was enabled. It can be simulated with two multiplication neurons, a sign neuron and two sum neurons. The figure 6.3 represents an extended memory neuron with several sequential runs. In this following document extended memory neurons will be used as single oracle neurons.



(a) The ANN

time	data	activation	Output
1	0	-1	0
2	5	-1	0
3	5	1	5
4	5	-1	5
5	7	-1	5
6	7	1	7
7	0	-1	7

(b) The runs

Figure 6.3: Extended memory

Definition 6.4. A *test* is a function $U^n \to \{-1, 1\}$, where T is the Herbrand universe and n is the arity of the test. A test that gives 1 succeeds. A test that gives -1 fails. The equality ($=$) and the inequality ($\neq$) are two tests of arity 2. The greater function ($>$) is a test that only applies to terms that represent integers.

A test can be computed by one or several neurons. In the case of simple tests like the equality, inequality or comparison, a simple usual neuron can be used. In the case of complex tests like '$T(x) = 1$ if x is composite, 0 otherwise' oracle neurons can be used. They are used in the technique 3.

To every test is associated a logic formula that is true for a given input arguments if an only if the test succeeds (return 1) for this input.

Definition 6.5. An *input predicate* is a predicate that is present at least once in the input of a training example.

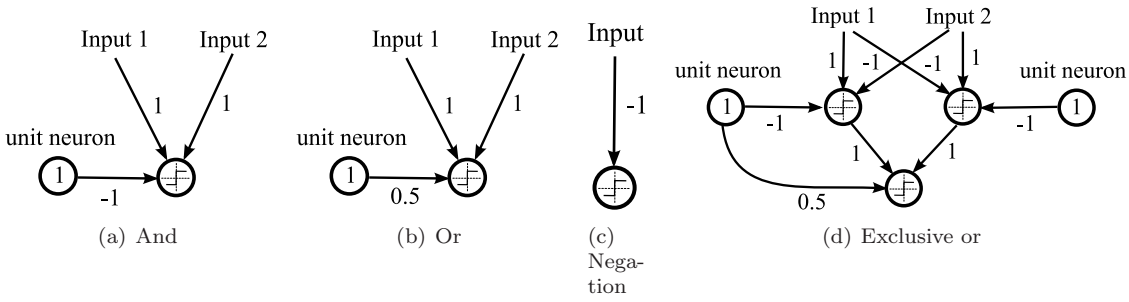
Definition 6.6. An *output predicate* is a predicate that is present at least once in the expected output of a training example.

6.2.1 Composition of tests

With the common logic operator ($\vee$,$\wedge$,$\neg$,xor,etc.) tests can be combined.

Each of them can be 'emulated' by one or several neurons. In the case of simple operator like $\vee$ or $\wedge$, a simple usual neuron can be used. In the case of complex tests like 'xor' oracle neurons can be used.

The figure 6.4 shows how a neuron can emulate the *or* ($\vee$), *and* ($\wedge$), *negation* ($\neg$) and *exclusive or* (xor) operators. In the next part of the document, this convention is called the *operators network convention*.



(a) And (b) Or (c) Negation (d) Exclusive or

Figure 6.4: Logic operators simulated by neurons

Definition 6.7. A test t_1 *subsumes* a test t_2 if an only if the associated formula of t_1 subsumes (logical implication) the associated formula of t_2.

Definition 6.8. A test t_1 is *equal* to a test t_2 if an only if t_1 subsumes t_2 and t_2 subsumes t_1 .

Example 6.2. the test $(a = b) \wedge (b = c)$ subsumes ($\models$) the test $(a = c)$.

6.2.2 Term typing

In order to simplify the research of hypotheses, a typing of terms is done. A type is associated to to every term. For every predicate, the type of all its arguments have to be given as input data of the algorithm. The term is defined by the predicate of the atom that contains it. For example, a predicate P of arity 2 may give the type 'natural' for its first argument and the type 'person' for its second argument. The typing does not go through functions. For example, if $f(a, b)$ has the type 'person', a and b will only get the type 'void' (default type).

This kind of typing does not allow to represent all kind of typing, but allow very important simplification of the network. A more expressive typing can be envisaged and is discussed in the conclusion of this report.

Two term of different type cannot be compared with equality of inequality test. Specialized functions and tests are only applied on terms with certain type. For example, the test greater ($<$) is a test of arity 2 between two natural numbers, and the test $length(L, N)$ ($length(L, N)$ fires if an only if L is a list of size N) is a test of arity 2 between a list and a natural number.

6.2.3 Parameters of the algorithm

In order to be used, several parameters have to be defined:

epsilon the learning rate used for the back propagation algorithm.

momentum the momentum used for the back propagation algorithm.

NoTraning the number of training used for the back propagation algorithm.

GenericANN the architecture for the activation ANN. For simple rule it can be a single neuron. For more complex rules, a small generic ANN is often enough. Example : Two layer with three neurons on the first layer and two on the second.

initDecompositionLevel the initial decomposition level of term. It expresses how many function decomposition the network will emulate. For example, to analyse the variable X in the term $f(g(X), a)$, the decomposition should be at least of 2.

initTestCompositionDepth the initial decomposition depth of test combinaison. It expresses a notion close to the depth of the binary conditions. For example $P \wedge Q$ is a condition of depth 1, and $P \wedge Q \wedge R$ ($= P \wedge (Q \wedge R)$) is a condition of depth 2.

initReplication the initial number of replication. It represents the maximum number of atoms with the same predicate in the rule that will be learned.

initGroundTerm if learned rules can contains ground terms.

initNoArgumentPattern the initial number of pattern created to infer the arguments of the outputs predicates. The higher this parameter is, the more complex the rule that can be infered. If all the output predicates have an arity of 0, this prameter is ignored.

initHeadTermGenerationDepth the initial maximum detph of creation of the outputs arguments. For example, the 'creation' of the term $g(f(a))$ form the term a needs a depth of creation of 2, but the creation of $g(f(a))$ from the term $f(a)$, only needs a depth of creation of 1. If all the output predicates have an arity of 0, this parameter is ignored.

initFunctionLevel the initial application depth of functions.

costDecompositionLevel the cost to increase the *DecompositionLevel* condition.

costGroundTerms the cost to set the *GroundTerms* condition.

costTestCompositionDepth the cost to increase the *TestCompositionDepth* condition.

costReplication the cost to increase the *Replication* condition.

costNoArgumentPattern the cost to increase the *NoArgumentPattern* condition.

costHeadTermGenerationDepth the cost to increase the *HeadTermGenerationDepth* condition.

costFunctionLevel the cost to increase the *FunctionLevel* condition.

maxANNTested the number of try to build the network. If a training fail, the language restrictions will be released according to their cost, and the training will start again.

beginningBuilding if **true**, the network is builded entirely at the beginning. In the other case, the network will be build during the training. For example, a neuron that combine two tests t_1 and t_2 will be created if both of those tests have at least succeed at once. If *beginningBuilding* is disabled, the final network is smaller and the training is (except for very small examples) faster.

allowAtomReplication if **true**, an atom given as input can be used several time in the same rule. For example, if this parameter is set to true, the atom $P(a, a)$ can trigger by its own the rule $P(X, Y) \wedge P(Y, X) \Rightarrow R$. If the parameter is set to false, a second instance of $P(a, a)$ is needed. Setting this parameter to false, increase the speed of the algorithm but induce some incompleteness problems.

6.2.4 The algorithm

Here is a formal description of the highest level of the technique 3 algorithm. The two subroutines *buildNetwork* and *enrich* are given in the next section.

Algorithm 16 Technique 3 : Induction on logic program through ANN Part 1/2

TrainingExample is the set of training example
$DecompositionLevel := initDecompositionLevel$
$TestCompositionDepth := initTestCompositionDepth$
$Replication := initReplication$
$GroundTerm := initGroundTerm$
$NoArgumentPattern := initNoArgumentPattern$
$HeadTermGenerationDepth := initHeadTermGenerationDepth$
for $j = 1$ to $maxANNTested$ **do**
  $succeed := true$
  $parameters := \{DecompositionLevel, TestCompositionDepth, Replication, GroundTerm, NoArgumentPattern, HeadTermGenerationDepth\}$
  Build the network $n = buildNetwork(parameters)$
  for $i = 1$ to $NoTraning + 1$ **do**
    for $ex = (Input, expectedOutput) \in$ **TrainingExample do**
      Reset all memory neurons
      for all the possible distribution of *Input* in the input layer of the network n **do**
        Load *Input* into the input layer according to the *input convention*.
        Make a run with the network n
      end for
      if beginningBuilding = false **then**
        Enrich the ANN n with $enrich(n, parameters)$
      end if
      Compute the error of the output layer with *expectedOutput* and the *output convention*.
      if $i = NoTraning + 1$ **then**
        if at least one of the output is bad according to the *output convention* **then**
          $succed := false$
        end if
      else
        Apply the back propagation algorithm to the network n
      end if
    end for
  end for

Algorithm 17 Technique 3 : Induction on logic program through ANN Part 2/2

  if $succed = true$ **then**
    break
  end if
  If j mod $costDecompositionLevel = 0$ **Then increase** DecompositionLevel
  If j mod $costTestCompositionDepth = 0$ **Then increase** TestCompositionDepth
  If j mod $costReplication = 0$ **Then increase** Replication
  If j mod $costGroundTerms = 0$ **Then set** GroundTerms
  If j mod $costNoArgumentPattern = 0$ **Then increase** NoArgumentPattern
  If j mod $costHeadTermGenerationDepth = 0$ **Then set** HeadTermGenerationDepth
end for
Return n

6.2.5 Building of the network

The ANN created by the *buildNetwork* function is composed of several layers. The following algorithm specify their definition. To help the understanding, each layer is explained individually. The figure 6.5 shows the interconnection between those layers.

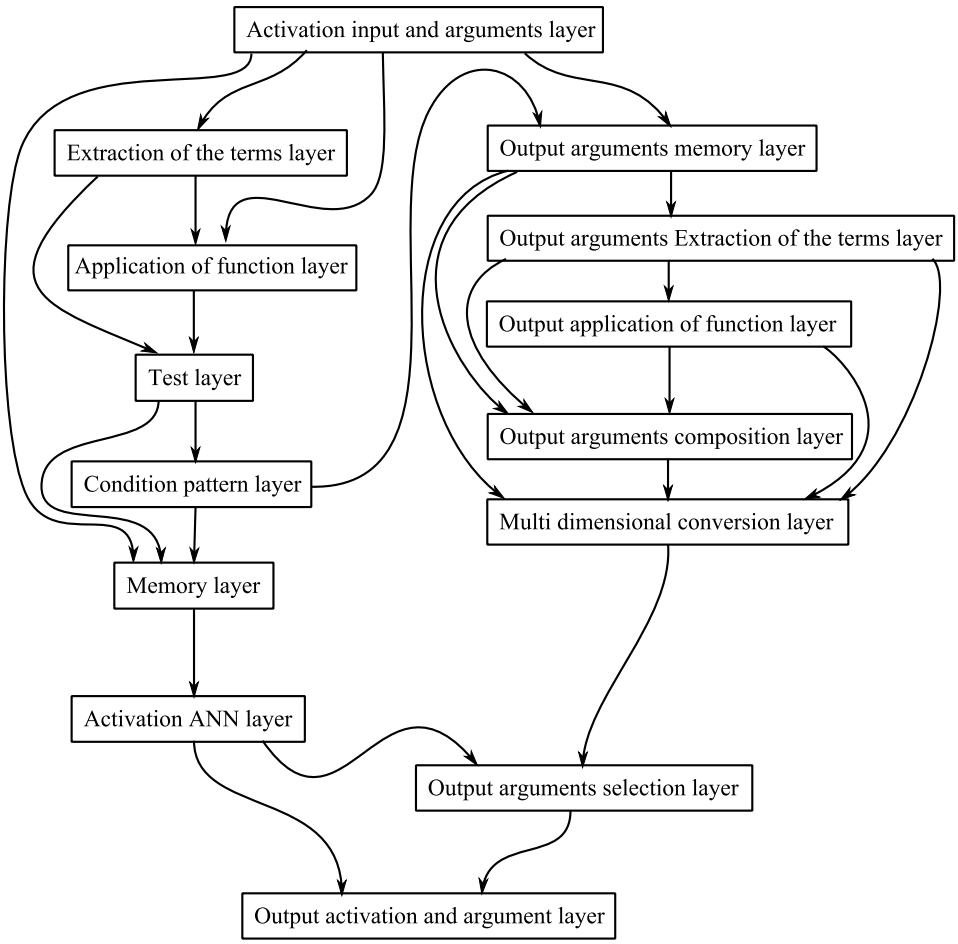


Figure 6.5: The interconnection between the layers

Algorithm 18 Building the Inductive Artificial Neural Network Part 1/14

procedure BUILDNETWORK

Test is the set of test that can be done on terms
(by default equality, difference and $<$ in integers)

Operator is the set of logical operator that can be used (by default $\vee$, $\wedge$, $\neg$)

Term is an empty set of condition and neuron associations (neuron,type,condition)

Condition is an empty set of condition, integer and neuron associations (neuron,depth,condition)

maxArityFunction is the maximum number of argument in a term function

inputArgumentLayer is an empty set of condition and neuron associations (neuron,type,condition)

memoryInputArgumentLayer is an empty set of condition and neuron associations (neuron,type,condition)

multiDimMemoryInputArgumentLayer is an empty set of condition and neuron associations (neuron,type,condition)

The activation input and arguments layer contains for every input predicate P time the number of replication, the *activation neuron Activation_P*, and for $i \in [1, arity(P)]$, an *argument neuron Argument_P_i*. It is trough this layer that input data (atoms) are loaded into the network.

Algorithm 19 Building the Inductive Artificial Neural Network Part 2/14

▷ Activation input and arguments layer

for every input predicate P_j **do**
  for $k = 1$ to *Replication* **do**
    Create the neuron $Activation\_P_j\_k$
    Add ($Activation\_P_j\_k$,0,P_j_k) to *Condition*
    for $l = 1$ to $arity(P_j)$ **do**
      Create the neuron $Argument\_P_j\_k\_l$
      Add ($Argument\_P_j\_k\_l$,*type*,P_j_k) to $Term$
      with *type* the type of the k^{th} argument of P
    end for
  end for
end for
set $inputArgumentLayer := Term$

The extraction of the terms layer contains extraction neurons over all the argument neurons. It decompose the arguments of the input atoms. For example, if the atom $P(a, f(b, g(c)))$ is presented to the *Activation input and arguments layer*. The *extraction of the terms layer* will generate the terms b, $g(c)$ and c from a and $f(b, g(c))$

Algorithm 20 Building the Inductive Artificial Neural Network Part 3/14

▷ Extraction of the terms layer

subterm := Term
for $i = 1$ to *DecompositionLevel* **do**
  savesubterm = subterm
  set $subterm = \emptyset$
  for $(p, t, c) \in savesubterm$ **do**
    if t is a composable type **then**
      Create the neuron n with the activation function E^0
      Connect p to n with a weigh of 1
      for all function F of the Herbrand **do**
        if *GroundTerm* **or** if the arity of F is greater or equal to one **then**
          Create the neuron m with the activation function *extended equality*
          Connect n to m with a weigh of 1
          Set the bias of m to $-Index(F)$
          Add (m,0,c) to *Condition*
        end if
      end for
      for $k = 1$ to $maxArityFunction$ **do**
        Create the neuron n with the activation function E^k
        Connect p to n with a weigh of 1
        Add (n,void,c) to $Term$
        Add (n,void,c) to *subterm*
      end for
    end if
  end for
end for

The application of function layer contains all the composition of function. For example, if the function $mult : (X, Y) \mapsto X \cdot Y$ on natural numbers is allowed. This layer will do all the computation of mult over the terms that represent natural numbers.

Algorithm 21 Building the Inductive Artificial Neural Network Part 4/14

▷ Application of function layer

```
subterm := Term
for i = 1 to FunctionLevel do
    savesubterm = subterm
    set subterm = ∅
    for all f ∈ Function do
        for all e_1 ∈ savesubterm do
            for all sub sets S = {e_2, ..., e_m} of element of Term do
                if for all the e_k = (n_k, t_k, c_k), t_k is the type of the k^th argument of f then
                    for all combinaison [(n_1, t_1, c_1), ..., (n_m, t_m, c_m − 1)] of element of S do
                        create a n neuron
                        define the function of n, and the connections of n_i to n according to the definition
of f.
                        Add (n,type,c_1 ∧ ... ∧ c_m) to Term, with type the type of the term produce by f
                        Add (n,type,c_1 ∧ ... ∧ c_m) to subterm, with type the type of the term produce by
f
                    end for
                end if
            end for
        end for
    end for
end for
```

Test layer contains tests neurons between all the argument neurons and the extracted terms. It does all the test between the arguments terms and the extracted terms. For example, if the only test is the equality and the only atom loaded in the system is $P(a, f(b, g(c)))$. The test between a, $f(b, g(c))$, b, $g(c)$ and c will be done.

Algorithm 22 Building the Inductive Artificial Neural Network Part 5/14

▷ Test layer

```
for all t ∈ Test do
    for all combinaison [(n_1, t_1, c_1), ..., (n_m, t_m, c_m)] of element of Term, with m the arity of the test t
do
        if the types t_k correspond to the type required by t then
            Create a neuron p of the test t
            Connect all the n_k to p with the weight required by the test t
            Add (p,0,c_1 ∧ ... ∧ c_m) to Condition
        end if
    end for
end for
```

The condition pattern layer contains condition neurons over all the test neurons and the activation neurons. It combines the different tests together. For example if the test $(W = X)$ and $(Y = Z)$ are combined together with the **and** operator, the resulting test will be $(W = X) \wedge (Y = Z)$. Since the number of test can be important, the parameter of the algorithm limits they number. If it appears that the learning cannot be done with this limitation, the training will start again with a less restrictive limit.

Algorithm 23 Building the Inductive Artificial Neural Network Part 6/14

▷ Condition pattern layer

if beginningBuilding=true **then**
  for operator $o \in Operator$ **do**
    ▷ Depending of the nature of the operator, the following operations can be done in a more efficient way
    for all combinaison $[(n_1, d_1, c_1), ..., (n_m, d_m, c_m)]$ of element of $Condition$ with m the arity of the operator o **do**
      set $newTest = (n_{newTest}, d_{newTest}, c_{newTest})$ with
      $n_{newTest} = NULL$
      $d_{newTest} = max(d_1, .., d_m) + 1$
      $c_{newTest} = o(c_1, ..., c_m)$
      if $d_{newTest} < TestCompositionDepth$ **and** $newTest$ is not equal to any test of $Condition$ **then**
        create a new neuron p according to the operator o neuron convention.
        set $n_{newTest} = p$
        connect all the $\{n_i\}_{i\in[1,m]}$ to the neuron p with the weights defined by the operator o neuron convention
        add $newTest$ to $Condition$
      end if
    end for
  end for
end if

The memory layer contains for every neuron of the *Condition pattern layer*, an associated memory neuron. it remember for every test, if it has been activated once.

Algorithm 24 Building the Inductive Artificial Neural Network Part 7/14

▷ Memory layer

for all $(n, c) \in Condition$ **do**
  Create a memory neuron m
  Connect n to m with a weight of 1
end for

The output arguments memory layer contains memory neurons that 'remember' arguments of the input atoms. For every test and test composition, the different arguments of the input atoms are remembered.

Algorithm 25 Building the Inductive Artificial Neural Network Part 8/14

▷ Output arguments memory layer

for all $(n_1, c_2) \in Condition$ **do**
  for all $(n_2, t, c_1) \in inputArgumentLayer$ **do**
    Create a extended memory neuron m
    Connect n_1 to m with a weigh of 1
    Connect n_2 to m with a weigh of 1
    add (m, t, c_1) to memoryInputArgumentLayer
  end for
end for

The output arguments Extraction of the terms layer contains extraction neurons over all the argument neurons coming from the *Output arguments memory layer*. it plays the same role as the *extraction of the terms layer* i.e. it decomposes terms.

Algorithm 26 Building the Inductive Artificial Neural Network Part 9/14

▷ Output arguments Extraction of the terms layer

```
subterm := memoryInputArgumentLayer
for i = 1 to DecompositionLevel do
    savesubterm = subterm
    set subterm = ∅
    for (p, t, c) ∈ savesubterm do
        if t is a composable type then
            for k = 1 to maxArityFunction do
                Create the neuron n with the activation function E^k
                Connect p to n with a weigh of 1
                Add (n,void,c) to memoryInputArgumentLayer
                Add (n,void,c) to subterm
            end for
        end if
    end for
end for
if GroundTerm then
    for all constants c do
        Create the neuron n with the activation function sum
        Connect the unit neuron to n with a weigh of E(index(c), 0)
        Add (n,void,c) to memoryInputArgumentLayer
        Add (n,void,c) to subterm
    end for
end if
```

The output application of function layer contains all the composition of function. For example, if the function $mult : (X, Y) \mapsto X \cdot Y$ on natural numbers is allowed. This layer will do all the computation of mult over the terms that represent natural numbers.

Algorithm 27 Building the Inductive Artificial Neural Network Part 10/14

▷ Output application of function layer

subterm := memoryInputArgumentLayer
for $i = 1$ to $FunctionLevel$ **do**
savesubterm = subterm
set $subterm = \emptyset$
for all $f \in Function$ **do**
for all $e_1 \in savesubterm$ **do**
for all sub sets $S = \{e_2, ..., e_m\}$ of element of $Term$ **do**
if for all the $e_k = (n_k, t_k, c_k)$, t_k is the type of the k^{th} argument of f **then**
for all combinaison $[(n_1, t_1, c_1), ..., (n_m, t_m, c_m - 1)]$ of element of S **do**
create a n neuron
define the function of n, and the connections of n_i to n according to the definition of f.
Add (n,*type*,$c_1 \wedge ... \wedge c_m$) to $memoryInputArgumentLayer$, with *type* the type of the term produce by f
Add (n,*type*,$c_1 \wedge ... \wedge c_m$) to $subterm$, with *type* the type of the term produce by f
end for
end if
end for
end for
end for
end for

The output arguments composition layer contains encoding neurons that compact terms into composed terms. For example, to compose a and $g(b)$ to $f(a, g(b))$.

Algorithm 28 Building the Inductive Artificial Neural Network Part 11/14

▷ Output arguments composition layer

subterm := memoryInputArgumentLayer
for $i = 1$ to $HeadTermGenerationDepth$ **do**
  savesubterm = subterm
  set $subterm = \emptyset$
  for all function F of the Herbrand with an arity m greater or equal to one **do**
    for all $e_1 \in savesubterm$ **do**
      for all sub sets $S = \{e_2, ..., e_m\}$ of element of $memoryInputArgumentLayer$ **do**
        if for all the $e_k = (n_k, t_k, c_k)$, t_k is a composable type **then**
          for all combinaison $[(n_1, t_1, c_1), ..., (n_m, t_m, c_m - 1)]$ of element of S **do**
            create $m + 1$ neurons $\{p_i\}_{i \in [0,m]}$ with the **Compose** activation function
            for $j = 1$ to m **do**
              connect n_i to p_i as the first connection with a weight of 1
              connect p_i to p_{i-1} as the second connection with a weight of 1
            end for
            connect the unit neuron to p_m as the second connection with a weight of 0
            connect the unit neuron to p_0 as the first connection with a weight of $index(F)$
            Add (p_0,void,$\top$) to $memoryInputArgumentLayer$
            Add (p_0,void,$\top$) to $subterm$
          end for
        end if
      end for
    end for
  end for
end for

The multi dimensional conversion layer contains neurons that convert from a single dimensional representation of terms to a multi dimensional representation. This multi dimensional representation is required to make the system converge. For example, the value $5 \in \mathbb{N}$ is convented to v_5, with $[v_i]_{i \in \mathbb{N}}$ is a base of a multi dimensional vectorial space.

Algorithm 29 Building the Inductive Artificial Neural Network Part 12/14

▷ Multi dimensional conversion layer

for all $(n, t, c) \in memoryInputArgumentLayer$ **do**
  create a neuron m with the **set multi dimension** activation function
  connect n to m
  add (m, t, c) to multiDimMemoryInputArgumentLayer
end for

The activation ANN layer contains a small generic ANN with as input, the memory layer. For example, a ANN with two layer, three neurons in the first layer, two on the second layer and all neurons of the first layer connected to the neurons of the second layer.

Algorithm 30 Building the Inductive Artificial Neural Network Part 13/13

▷ Activation ANN layer

Create the *activation layer* according to the architecture definition
Connect all the single dimensional memory neurons to all the neurons of the first layer of the *activation layer*
Set those connection to cut back propagation

The output arguments selection layer contains neurons that select the arguments of the output predicates. This layer will for example, if the learning rule is $P(X,Y) \Rightarrow Q(Y)$, that the first argument of the output predicate Q is the second argument of the input predicate P, in this particular rule. But the selection can be more complex like with rule of the kind $P(f(X), g(Y)) \Rightarrow Q(h(X,Y))$

Algorithm 31 Building the Inductive Artificial Neural Network Part 14/14

▷ Output arguments selection layer

for every output predicate Q_i **do**
  for $j = 1$ to $arity(Q_i)$ **do**
    for $k = 1$ to $NoArgumentPattern$ **do**
      create a neuron $Argument\_Q_i\_j\_k$ with the **multidimentionnal sum** activation function
      Connect every neuron of the last layer of the *Activation ANN layer* to $Argument\_Q_i\_j\_k$ with a connection restricted to weight with a positive value
      for all $(n,t,c) \in multiDimMemoryInputArgumentLayer$ **do**
        if t is the same type as the j^{th} argument of the predicate Q_i **then**
          connect n to $Argument\_Q_i\_j\_k$ with a connection restricted to weight with a positive value
        end if
      end for
    end for
  end for
end for

The output activation and argument layer contains for every output predicate Q, the *activation neuron* $Activation\_Q$ connected from the *Activation ANN layer*. It is the final layer were the results are presented i.e. the outputs atoms.

Algorithm 32 Building the Inductive Artificial Neural Network Part 13/14

▷ Output activation and argument layer

for every output predicate Q_i **do**
  Create the neuron $Activation\_Q_i$ with the **single dimention sum** activation function
  Connect every neuron of the last layer of the *Activation ANN layer* to $Activation\_Q_i$
  for $j = 1$ to $arity(Q_i)$ **do**
    create a neuron $Argument\_Q_i\_j$ with the **set single dimention** activation function
    for $k = 1$ to $NoArgumentPattern$ **do**
      connect $Argument\_Q_i\_j\_k$ to $Argument\_Q_i\_j$ with a connection restricted to weight with a positive value
    end for
  end for
end for
end procedure

The *enrich* sub routine build a small part of the network according to the currently activated *tests neurons*.

Algorithm 33 Enrich an Inductive Artificial Neural Network 1/4

procedure ENRICH(DecompositionLevel,TestCompositionDepth,Replication,GroundTerm)
Condition is the set of condition, integer and neuron associations (neuron,depth,condition) created during the network building
Activated is an empty set of condition, integer and neuron associations (neuron,depth,condition)
for all $(n, d, c) \in Condition$ **do**
if the neuron n is activated i.e. the value of n is greater than 0 **then**
add (n, d, c) to $Activated$
end if
end for
$newNeuronToAdd := true$
while $newNeuronToAdd = true$ **do**
$newNeuronToAdd := false$
set temporaryMemoryInputArgumentLayer := $\emptyset$
set temporaryConditions := $\emptyset$
for operator $o \in Operator$ **do**
▷ Depending of the nature of the operator, the following operations can be done in a more efficient way
for all combinaison $[(n_1, d_1, c_1), ..., (n_m, d_m, c_m)]$ of element of $Activated$
with m the arity of the operator o **do**
set $newTest = (n_{newTest}, d_{newTest}, c_{newTest})$ with
$n_{newTest} = NULL$
$d_{newTest} = max(d_1, .., d_m) + 1$
$c_{newTest} = o(c_1, ..., c_m)$
if $d_{newTest} < TestCompositionDepth$ **and** $newTest$ is not equal to any test of $Condition$ **then**
create a new neuron p according to the operator o neuron convention.
set $n_{newTest} = p$
connect all the $\{n_i\}_{i \in [1,m]}$ to the neuron p with the weights defined by the operator o neuron convention
add $newTest$ to $Condition$
add $newTest$ to $Activated$
add $newTest$ to $temporaryConditions$
$newNeuronToAdd := true$
end if
end for
end for

Algorithm 34 Enrich an Inductive Artificial Neural Network 2/4

```
if newNeuronToAdd==true then
    for all (n_1, c_2) ∈ temporaryConditions do
        for all (n_2, t, c_1) ∈ inputArgumentLayer do
            Create a extended memory neuron m
            Connect n_newTest to m with a weigh of 1
            Connect n_2 to m with a weigh of 1
            add (m, t, c_1) to memoryInputArgumentLayer
            add (m, t, c_1) to temporaryMemoryInputArgumentLayer
        end for
    end for
    subterm := temporaryMemoryInputArgumentLayer
    for i = 1 to DecompositionLevel do
        savesubterm = subterm
        set subterm = ∅
        for (p, t, c) ∈ savesubterm do
            if t is a composable type then
                for k = 1 to maxArityFunction do
                    Create the neuron n with the activation function E^k
                    Connect p to n with a weigh of 1
                    Add (n,void,c) to memoryInputArgumentLayer
                    Add (n,void,c) to temporaryMemoryInputArgumentLayer
                    Add (n,void,c) to subterm
                end for
            end if
        end for
    end for
```

Algorithm 35 Enrich an Inductive Artificial Neural Network 3/4

subterm := temporaryMemoryInputArgumentLayer
for $i = 1$ to $HeadTermGenerationDepth$ **do**
savesubterm = subterm
set $subterm = \emptyset$
for all function F of the Herbrand with an arity m greater or equal to one **do**
for all $e_1 \in savesubterm$ **do**
for all sub sets $S = \{e_2, ..., e_m\}$ of element of $memoryInputArgumentLayer$ **do**
if for all the $e_k = (n_k, t_k, c_k)$, t_k is a composable type **then**
for all combinaison $[(n_1, t_1, c_1), ..., (n_m, t_m, c_m - 1)]$ of element of S **do**
create $m + 1$ neurons $\{p_i\}_{i \in [0,m]}$ with the **Compose** activation function
for $j = 1$ to m **do**
connect n_i to p_i as the first connection with a weight of 1
connect p_i to p_{i-1} as the second connection with a weight of 1
end for
connect the unit neuron to p_m as the second connection with a weight of 0
connect the unit neuron to p_0 as the first connection with a weight of $index(F)$
Add (p_0,void,$\top$) to $memoryInputArgumentLayer$
Add (p_0,void,$\top$) to $temporaryMemoryInputArgumentLayer$
Add (p_0,void,$\top$) to $subterm$
end for
end if
end for
end for
end for
set temporaryMultiDimMemoryInputArgumentLayer := $\emptyset$
for all $(n, t, c) \in temporaryMemoryInputArgumentLayer$ **do**
create a neuron m with the **set multi dimension** activation function
connect n to m
add (m, t, c) to multiDimMemoryInputArgumentLayer
add (m, t, c) to temporaryMultiDimMemoryInputArgumentLayer
end for

Algorithm 36 Enrich an Inductive Artificial Neural Network 4/4

```
            for every output predicate Q_i do
                for j = 1 to arity(Q_i) do
                    for k = 1 to NoArgumentPattern do
                        for all (n, t, c) ∈ temporaryMultiDimMemoryInputArgumentLayer do
                            if t is the same type as the j^th argument of the predicate Q_i then
                                connect n to Argument_Q_i_j_k with a connection restricted to weight with a
positive value
                            end if
                        end for
                    end for
                end for
            end for
        end if
    end while
end procedure
```

6.3 Detailed instances of problem

6.4 Instance 1

6.4.1 Creation of the network

The following section presents an detailed example of run of this technique of the training example given in the table 6.1. The expected extracted rule is $P(X,Y) \wedge Q(Y) \Rightarrow R$.

$P(a,b) \wedge Q(b) \Rightarrow R$
$P(a,b) \wedge Q(c) \Rightarrow$
$P(b,a) \wedge Q(a) \Rightarrow R$
$P(c,b) \wedge Q(a) \Rightarrow$

Table 6.1: Training example

The parameter of the algorithm are set to the following values:

epsilon = 0.1.

momentum = 0.1.

NoTraning = 1000.

GenericANN = a two layer, with two neuron by layer ANN network.

initDecompositionLevel = 0, since the example does not use term function, the decomposition is useless.

initTestCompositionDepth = 1. This value will be increased if it is not enough. But for this particular example, it is enough.

initReplication = 1, since there is no need of replication. This value will be increased if it is not enough. But for this particular example, it is enough.

initGroundTerm = 0, since there is no need of ground terms in the body of the rule. This value will be change if it is not working without it. But for this particular example, it is good without it.

initNoArgumentPattern $= 1$, since there is only on rule, there is only one pattern of rule (The contrapositive is not true). More of that, in this case, the output term (R) does not have argument. Therefore this parameter does not have any effect.

initHeadTermGenerationDepth $= 0$, since there is not function on the head of the rule (The contrapositive is not true).

costDecompositionLevel $=$. Since the training will succeed with the initial value. The costs will not be used.

costGroundTerms $= -1$. Since the training will succeed with the initial value. The costs will not be used.

costTestCompositionDepth $= -1$. Since the training will succeed with the initial value. The costs will not be used.

costReplication $= -1$. Since the training will succeed with the initial value. The costs will not be used.

costNoArgumentPattern $= -1$. Since the training will succeed with the initial value. The costs will not be used.

costHeadTermGenerationDepth $= -1$. Since the training will succeed with the initial value. The costs will not be used.

maxANNTested $= 1$. Since the initial parameters are good for the rule to extract.

beginningBuilding $= \mathit{true}$. The network is small enough to be completely build at the begining.

The only *operator* used is the *and* operator. The only *test* used is the *equality* test. There is not typing of the variable.

The following inference network is generated.

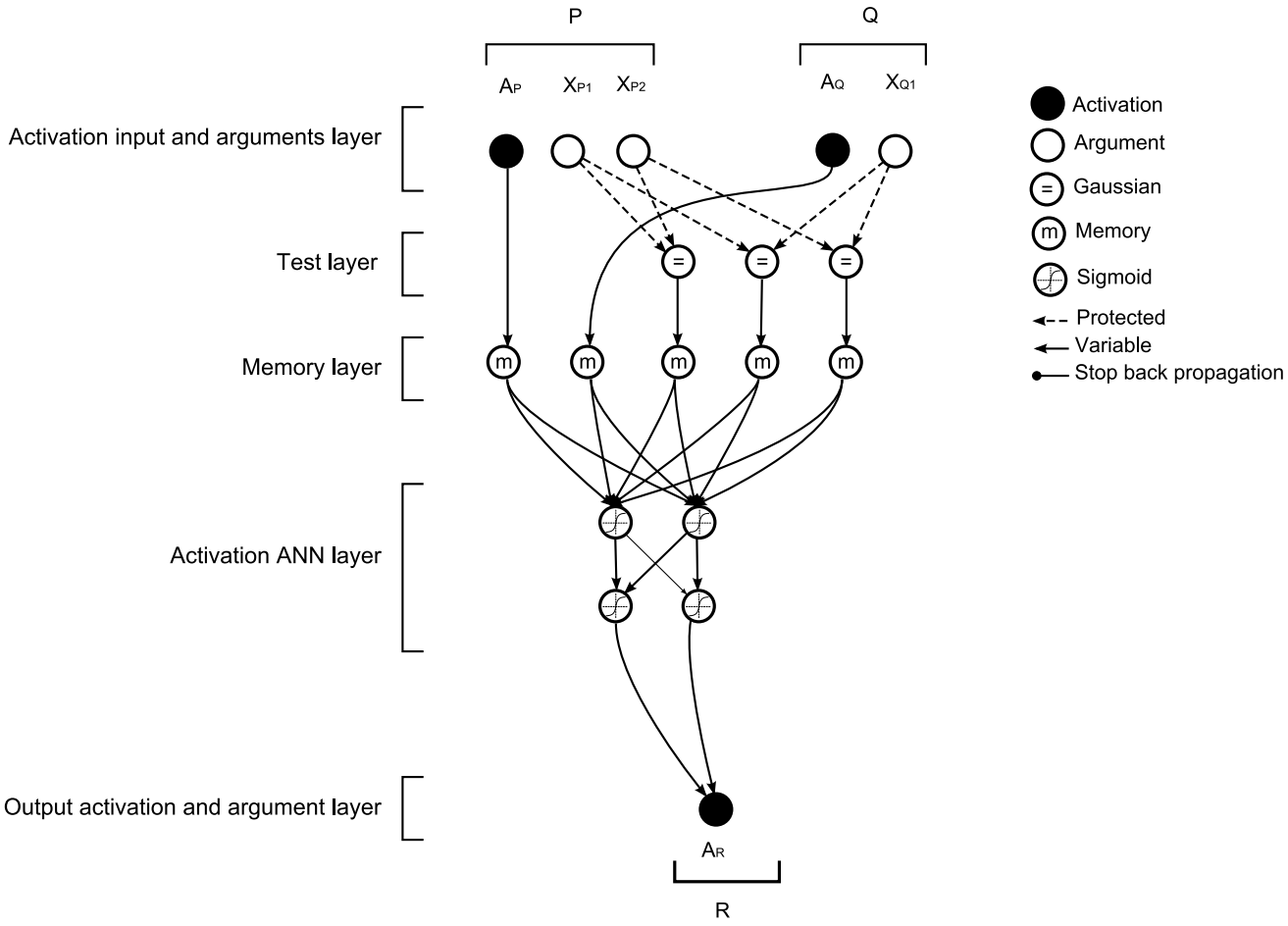


Figure 6.6: The inference network

Since the output term (R) does not have argument, all the layers dedicated to produce the output arguments are empty (Output arguments memory layer,Output arguments Extraction of the terms layer,Output arguments composition layer,Multi dimensional conversion layer,Output arguments selection layer). Since there is not term extraction ($initDecompositionLevel = 0$), the *Extraction of the terms layer* is empty. Since there is not composition of condition ($initTestCompositionDepth = 1$), the *Condition pattern layer* is empty.

6.4.2 Training of the network

For each of the four training examples of this instance of problem, there is only one possible distribution of the input atoms in the *Activation input and arguments layer*.

Let's consider the first training example $P(a, b) \wedge Q(b) \Rightarrow R$. This example contains only one instance of atom with the term P and only one instance of atom with the term Q. Since there is not replication of the input atoms ($initReplication = 1$), the only distribution is to bond $P(a, b)$ to the P_1 input and $Q(b)$ to the Q_1 input.

For this first training example, the input of the ANN is given by the table 6.2. Let's suppose $index(a) = 1$, $index(b) = 2$ and $index(c) = 3$.

$$
\begin{array}{l}
value(A_p) = 1 \\
value(X_{p_1}) = encode(a) = E(index(a), 0) = 2 \\
value(X_{p_2}) = encode(b) = E(index(b), 0) = 5 \\
value(A_q) = 1 \\
value(X_{q_1}) = encode(a) = E(index(a), 0) = 5
\end{array}
$$

Table 6.2: Input value of the ANN for the first training example $P(a, b) \wedge Q(b) \Rightarrow R$

The expected output is the atom R, i.e. $value(A_r) = 1$. The back propagation error of A_r is therefore $error(A_r) = 1 - value(A_r)$.
The following figure present the value of the memory layer after the first "loading" of data.

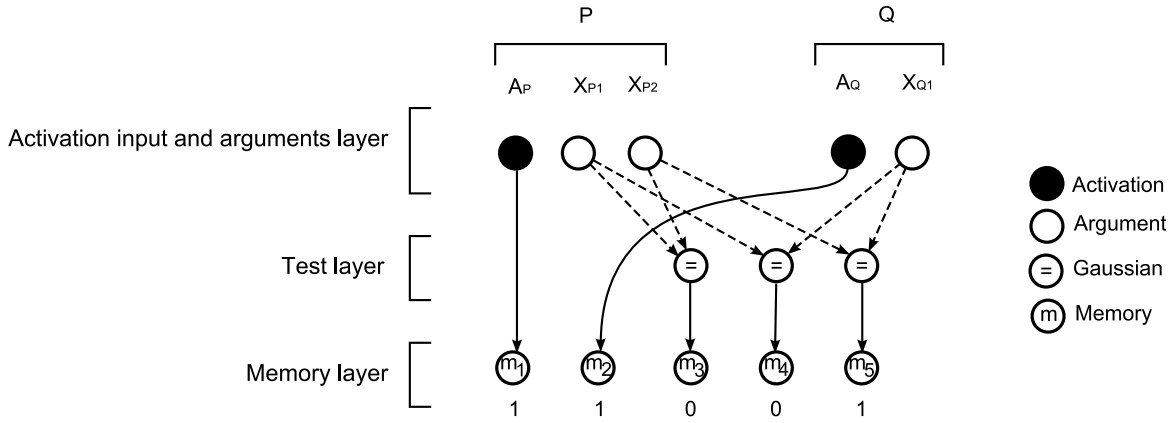


Figure 6.7: The inference network

The meaning of the different memory neuron is the following one:

$m_1 = 1$ means that at least one occurrence of P exist.

$m_2 = 1$ means that at least one occurrence of Q exist.

$m_3 = 0$ means that there is not occurrence $P(X, Y)$ with $X = Y$.

$m_4 = 0$ means that there is not occurrence $P(X, Y)$ and $Q(Z)$ with $X = Z$.

$m_5 = 1$ means that there is at least one occurrence $P(X, Y)$ and $Q(Z)$ with $Y = Z$.

6.5 Instance 2

6.5.1 Creation of the network

The following section presents an detailed example of run of this technique of the training example given in the table 6.3. The expected extracted rule is $P(X, Y) \wedge Q(f(Y)) \Rightarrow R(g(X))$.

$$
\begin{array}{l}
\hline
P(b) \wedge Q(f(b)) \Rightarrow R(g(b)) \\
P(b) \wedge Q(c) \Rightarrow \\
P(b) \wedge Q(f(c)) \Rightarrow \\
P(a) \wedge Q(f(b)) \Rightarrow \\
P(a) \wedge Q(f(a)) \Rightarrow R(g(a)) \\
P(a) \wedge Q(g(a)) \Rightarrow
\end{array}
$$

Table 6.3: Training example

This example shows several new features that the system can handle:

The generation of argument for the output predicate (R has one argument).

The capacity to handle functions (f) in the body of the rules.

The capacity to generate functions (g) in the head of the rules i.e. for example, for the first example, the term $g(b)$ has to be constructed since it is not present anywhere else. It would not have been the case if the argument of R was $f(a)$.

The parameter of the algorithm are set to the following values:

epsilon = 0.1.

momentum = 0.1.

NoTraning = 1000.

GenericANN = a two layer, with two neuron by layer ANN network.

initDecompositionLevel = 1, the rule's body contains function with a maxima depth of 1.

initTestCompositionDepth = 2. This value will be increased if it is not enough. But for this particular example, it is enough.

initReplication = 1, since there is no need of replication. This value will be increased if it is not enough. But for this particular example, it is enough.

initGroundTerm = 0, since there is no need of ground terms in the body of the rule. This value will be change if it is not working without it. But for this particular example, it is good without it.

initNoArgumentPattern = 2, the number of pattern for the output argument of R. A value of 1 would have been enough.

initHeadTermGenerationDepth = 1, the rule's head contains function with a maxima depth of 1.

costDecompositionLevel =. Since the training will succeed with the initial value. The costs will not be used.

costGroundTerms = -1. Since the training will succeed with the initial value. The costs will not be used.

costTestCompositionDepth = -1. Since the training will succeed with the initial value. The costs will not be used.

costReplication = -1. Since the training will succeed with the initial value. The costs will not be used.

costNoArgumentPattern = -1. Since the training will succeed with the initial value. The costs will not be used.

costHeadTermGenerationDepth = -1. Since the training will succeed with the initial value. The costs will not be used.

maxANNTested = 1. Since the initial parameters are good for the rule to extract.

beginningBuilding = *true*. The network is small enough to be completly build at the begening.

The induction network is too complex to be represented completely. Therefore, a simplified representation is presented. The inference network is presented in the figure 6.8.

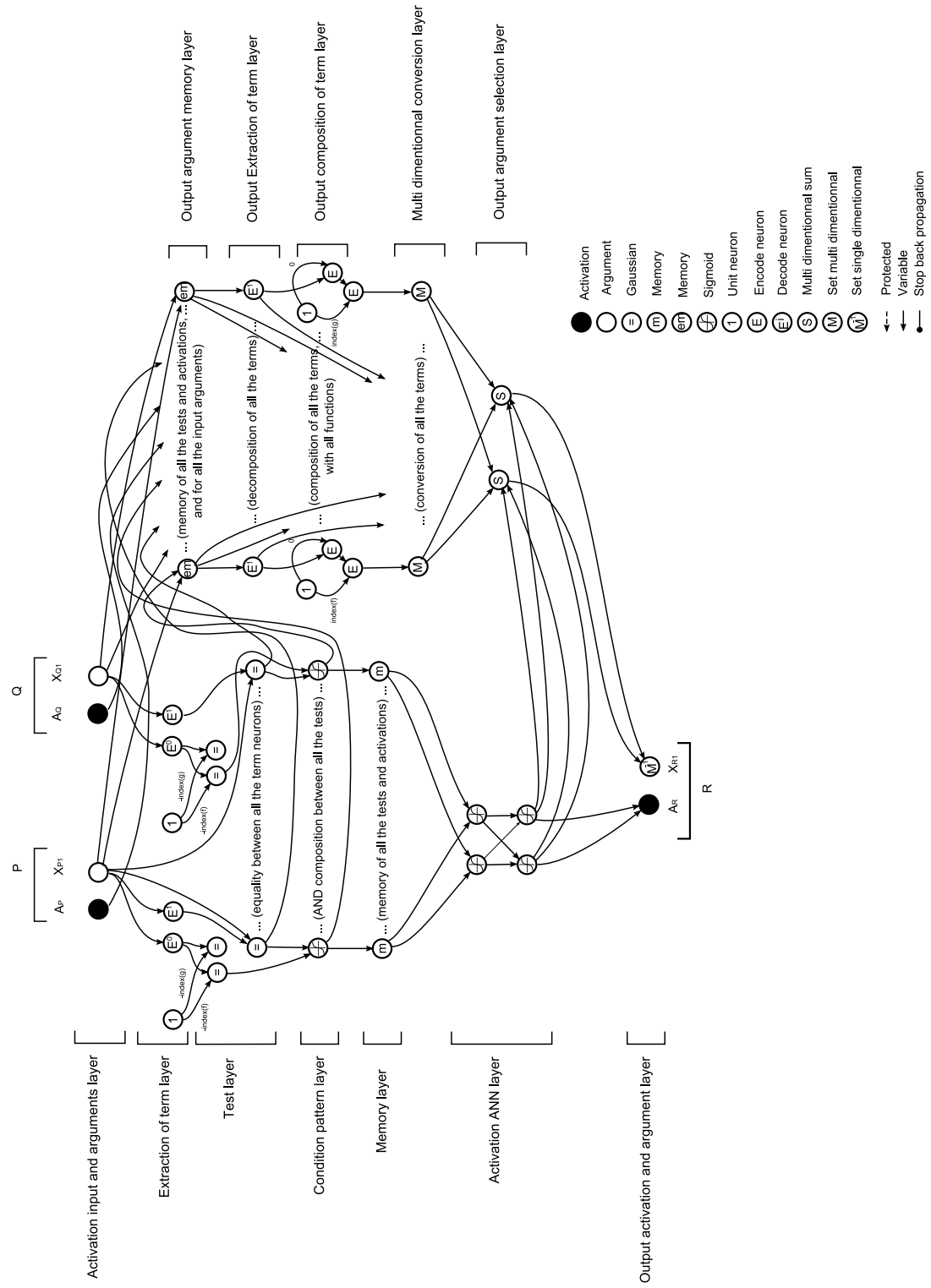


Figure 6.8: The inference network

6.6 Examples of run

6.6.1 Simples runs

All the problems that were correctly handled by the previous techniques are correctly handled by this technique. To show and emphasis on some of the features, some new simple example of run presented here.

The example 1 is a minimal example. The good rule is $P(X) \Rightarrow Q(X)$.

(a) Training set

$P(a) \Rightarrow Q(a)$
$P(b) \Rightarrow Q(b)$
$P(c) \Rightarrow Q(c)$

(b) Evaluation set

$P(d) \Rightarrow Q(d)$
$P(e) \Rightarrow Q(e)$
$P(f) \Rightarrow Q(f)$

(c) Result

Characteristics	Value
Success rate	100%
Training time	$0.23s$
Number of input neurons	3
Number of hidden neurons	5
Number of output neurons	2

Table 6.4: Example 1

The example 2 shows a simple argument selection. The good rule is $P(X, Y) \Rightarrow R(Y)$.

(a) Training set

$P(a, a) \Rightarrow R(a)$
$P(a, b) \Rightarrow R(b)$
$P(a, c) \Rightarrow R(c)$
$P(b, c) \Rightarrow R(c)$
$P(b, a) \Rightarrow R(a)$

(b) Evaluation set

$P(m, n) \Rightarrow R(n)$
$P(m, o) \Rightarrow R(o)$

(c) Result

Characteristics	Value
Success rate	100%
Training time	$0.21s$
Number of input neurons	4
Number of hidden neurons	13
Number of output neurons	2

Table 6.5: Example 2

The example 3 uses the decomposition of functions. The good rule is $P(X, f(X)) \wedge Q(g(X, f(X))) \Rightarrow R$.

(a) Training set

$P(a, f(a)) \wedge Q(g(a, f(a))) \Rightarrow R$
$P(a, f(b)) \wedge Q(g(a, f(a))) \Rightarrow$
$P(a, f(a)) \wedge Q(c) \Rightarrow$
$P(a, f(a)) \wedge Q(g(c, f(a))) \Rightarrow$
$P(a, f(a)) \wedge Q(g(a, f(d))) \Rightarrow$
$P(a, f(a)) \Rightarrow$
$Q(g(a, f(a))) \Rightarrow$

(b) Evaluation set

$P(d, f(d)) \wedge Q(g(d, f(d))) \Rightarrow R$
$P(d, f(e)) \wedge Q(g(d, f(d))) \Rightarrow$
$P(d, f(d)) \wedge Q(h) \Rightarrow$
$P(d, f(d)) \wedge Q(g(h, f(d))) \Rightarrow$
$P(d, f(d)) \wedge Q(g(d, f(e))) \Rightarrow$
$P(d, f(d)) \Rightarrow$
$Q(g(d, f(d))) \Rightarrow$

(c) Result

Characteristics	Value
Success rate	100%
Training time	$0.27s$
Number of input neurons	6
Number of hidden neurons	100
Number of output neurons	1

Table 6.6: Example 3

The input of the example 4 contains a lot of irrelevant data. The difficulty for the system is to select the good information. The good rule is $P(X, X) \vee Q(Y, Y) \Rightarrow R$.

(a) Training set

$P(a, a) \wedge Q(b, b) \wedge Q(a, b) \wedge P(a, b) \wedge P(b, a) \Rightarrow R$
$Q(b, b) \wedge Q(a, b) \wedge P(a, b) \wedge P(b, a) \Rightarrow$
$P(a, a) \wedge Q(a, b) \wedge P(a, b) \wedge P(b, a) \Rightarrow$

(b) Evaluation set

$P(c, c) \wedge Q(d, d) \wedge Q(c, d) \wedge P(c, d) \wedge P(d, c) \Rightarrow R$
$Q(d, d) \wedge Q(c, d) \wedge P(c, d) \wedge P(d, c) \Rightarrow$
$P(c, c) \wedge Q(c, d) \wedge P(c, d) \wedge P(d, c) \Rightarrow$

(c) Result

Characteristics	Value
Success rate	100%
Training time	$0.21s$
Number of input neurons	7
Number of hidden neurons	15
Number of output neurons	1

Table 6.7: Example 4

6.6.2 Michalski's train problem

A N fold test have been done with the Michalski's train problem. At every run, one of the train is excluded from the training data, and put back for the evaluation.

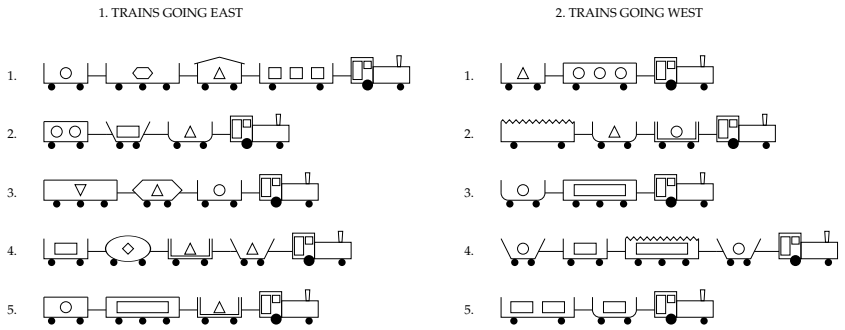


Figure 6.9: Michalski's train problem

Definition 6.9. The Michalski's train problem is a binary classification problem. The data set is composed of ten trains with different features (number of car, size of the cars, shape of the cars, object in the cars, etc.). Five of the train are going to the East, and the five other are going to the West. The problem is to found the relation between the features of the trains and theirs destination.

For every fold, with the parameter given bellow, the third technique produces a network with the following characteristics in in average of 36 seconds. All the runs show a 100% succes rate. The global result is therefore a succes rate of 100%.

In order to reduce the size of the network, the typing of the terms is done. 5 types are defined (car,shape,number,shape2,numbe
The association of the types and the predicate is presented in the table 6.8.

Short(car)
Closed(car)
Long(car)
Open(car)
Infront(car,car)
Shape(car,shape)
Load(car,shape2,number)
Wheels(car,number2)
Double(car)
Jagged(car)

Table 6.8: Types of the predicates arguments

Parameter	Value
epsilon	0.1
momentum	0.1
NoTraning	1000
GenericANN	a two layer, with two neuron by layer ANN network.
initDecompositionLevel	0
initTestCompositionDepth	2
initReplication	1
initGroundTerm	0
initNoArgumentPattern	2
initHeadTermGenerationDepth	1
costDecompositionLevel	-1
costGroundTerms	-1
costTestCompositionDepth	-1
costReplication	-1
costNoArgumentPattern	-1
costHeadTermGenerationDepth	-1
maxANNTested	5
beginningBuilding	*true*

Table 6.9: Parameter of the algorithm

Characteristics	Value
Number of input neurons	26
Number of hidden neurons	2591
Number of output neurons	1

Table 6.10: Characteristics of the induction network

6.6.3 Parity problem

The following example learn the notion of parity (oddness or evenness) of a natural number. The common way to represent this concept in logic is presented in the following logic program.

$$\begin{aligned} even(0) \\ odd(1) \\ even(s(X)) &\leftarrow odd(X) \\ odd(s(X)) &\leftarrow even(X) \end{aligned}$$

To prove the oddness or the evenness of a number greater than 1, the rules have to be used more than once. The techniques developed in this project are based on direct consequence operators i.e. they are not able to apply recursively a same rule. Therefore, this representation of parity is not well suited.

There is several other ways to represent the parity concept with non recursive rules.

$$even(X) \leftarrow (\left\lfloor \frac{X}{2} \right\rfloor \cdot 2 = X)$$
$$even(X) \leftarrow (\left\lfloor \frac{X}{2} \right\rfloor = \left\lfloor \frac{X+1}{2} \right\rfloor)$$
$$odd(X) \leftarrow \neg even(X)$$
$$...$$

Therefore, with if the correct functions and test are allowed, the Technique 3 can learn the concept of parity. In this following run, the system learn the notion of parity from the following example. The argument of the predicate $Number(\_)$ is typed as a $Natural$ ($\mathbb{N}$).

$$Number(0) \Rightarrow Even.$$
$$Number(1) \Rightarrow .$$
$$Number(2) \Rightarrow Even.$$
$$Number(3) \Rightarrow .$$
$$Number(10) \Rightarrow Even.$$
$$Number(15) \Rightarrow .$$
$$Number(18) \Rightarrow Even.$$
$$Number(35) \Rightarrow .$$

The trained ANN is tested on all numbers from 0 to 100.

Characteristics	Value
Success rate	100%
Training time	$0.23s$
Number of input neurons	3
Number of hidden neurons	94
Number of output neurons	1

Table 6.11: Parity result

Chapter 7

Comparison of Induction techniques with other inductive techniques

Except for some specialized domains, more of the scientific domains can be described with the first order logic semantic. Therefore Induction on first order logic is a very power tool with a lot of applications. However, because of its complex nature, there is currently not ultimate technique of induction on first order logic.

S. Muggleton is developing since 1991 an induction technique (ILP) [**?**] for the first order logic based in the *Inverse Entailment* (see chapter 1.4). This technique is directly operating on first order logic programs.

Artur S. d'Avila Garcez exposed in 2009 a new technique of induction on first order logic through artificial neural network [**?**]. ANNs are strong against noisy data, they are easily 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.

Like M. d'Avila Garcez's relational technique, the solutions I developed are based on artificial neural networks. However, the approach I followed for the development of those techniques differ in the following way:

First of all, ANNs are often considered as a 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 node for each of those layers. By opposition, the approach I used was to very carefully define the internal structure of the network in order to be closely connected to the first order logic semantic. The outcomes are the following ones:

A smaller network.

A direct way to extract learned rule from the network.

A good control of the convergence of the network.

The capacity to specify the king of rule I wanted to learn (space and time optimization, better generalization of the examples, limitation of the over learning).

The ability to deal with different kind of data inside the same network (activation level and terms).

The last point is a key of an interesting feature. Like S. Muggleton's Inverse Entailment technique, and by opposition to M. d'Avila Garcez's technique, the system described in this report is able to *generate* terms, and not only to test them. For example, to test if the answer of a problem is $P(a)$, M. d'Avila Garcez's relational technique needs to test the predicate P on every possible term i.e. $P(a)$, $P(b)$, $P(c)$, $P(f(a))$, etc. In the general case of infinite Herbrand universe, this operation takes an infinite time. The approach followed in the techniques described in this report directly generate the output argument. Therefore, we have the guaranty that the output argument is unique and will be available in a finite time.

Chapter 8

Conclusion and extension of this work

The work presented in this report was done during three months. Therefore, some points have not been completely explored, and some others have not even been considered. This chapter discusses some of those points.

8.1 More powerful term typing

The typing of term used in the two last techniques is defined through the predicates. For example, the predicate P with an arity of two, can accept as first argument a term of type *natural*, and as second argument a term of type *void* (the more general type). More of that, for example, whatever is the type of $f(a, g(b))$, the types of a and $g(b)$ are *void*. The typing is encoded architecturally, and it directly helps to reduce the size of the network.

However, this restriction on the typing may be inappropriate, and more general typing may be interesting to study. For example, by creating a special predicate $object(X, Y)$, with X a given term, and Y the type of X. By opposition to the current typing, this new kind of typing would not decrease the complexity of the network, but may help the convergence thank to proper typing restrictions.

This typing may be used conjointly with the already developed typing.

8.2 Increasing of the rule's power

The rules that were learned are restricted to the first order logic rules. However, more expressive rules can be learned with appropriate network architecture. For example, the extension of the notion of equality to the function can allow dealing with rules containing Meta functions.

For example, rule of the kind of $P([F](X), [G](X)) \wedge ([F] \neq [G]) \Rightarrow R(X)$, with $[F]$ and $[G]$ two meta-functions.

In the same spirit, rules and induction over functions over functions can be considered.

For example, rule of the kind of $P([F](X), [G](X)) \wedge (\Psi([F]) = [G]) \Rightarrow R(X)$, with $[F]$ and $[G]$ two meta-functions, and Ψ a function over functions.

8.3 Extraction of the rules

The first order semantic of the rule encoded in the ANN is in direct correspondence with the network architecture. Therefore, the extraction of the rule can be done directly by the analyze of the weights of some nodes of the network.

The following example shows how to extract a simple rule.
Suppose the following training set:

$P(a,a) \wedge Q(b,b) \wedge Q(a,b) \wedge P(a,b) \wedge P(b,a) \Rightarrow R$
$Q(b,b) \wedge Q(b,a) \wedge P(a,b) \wedge P(b,a) \Rightarrow R$
$P(c,c) \wedge Q(b,a) \wedge Q(a,b) \wedge P(b,a) \Rightarrow R$
$Q(b,a) \wedge Q(a,b) \wedge P(b,a) \Rightarrow$
$Q(a,b) \wedge P(a,b) \wedge P(b,a) \Rightarrow$

Table 8.1: Training set

The computed ANN is the following one.

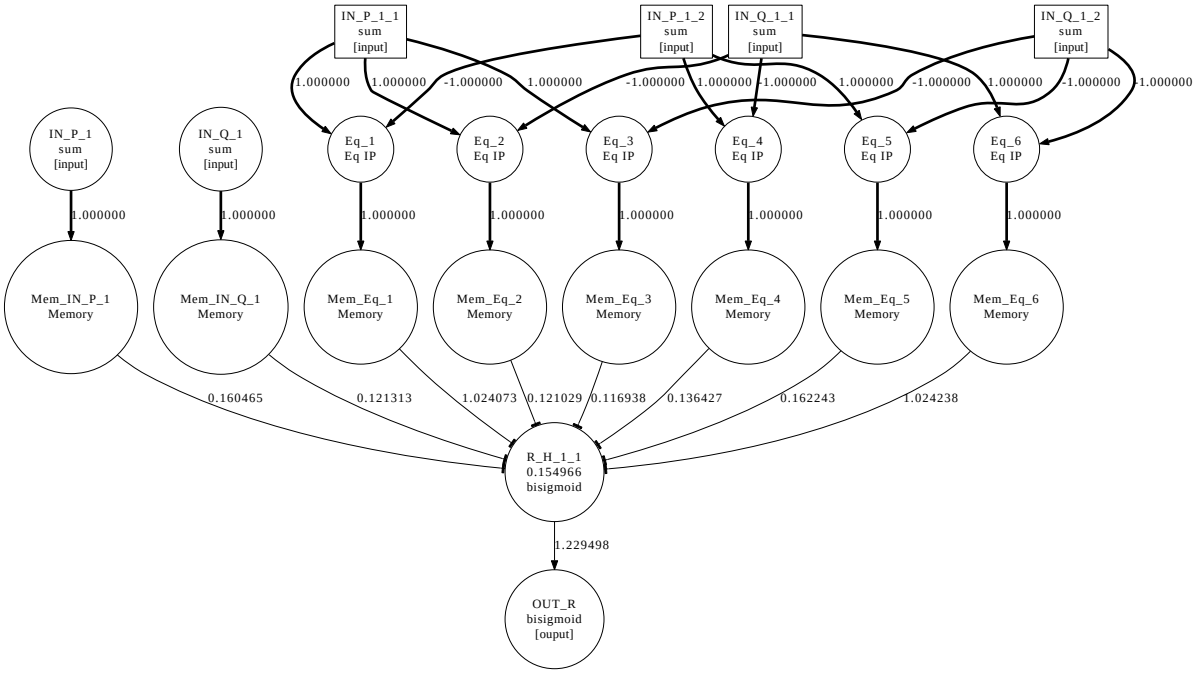


Figure 8.1: The produced ANN

The important connection to analyze are the ones that finish on the neuron $R\_H\_1\_1$. Informally, Out_R fires if $R\_H\_1\_1$ fires. And $R\_H\_1\_1$ fires if Mem_Eq_1 or Mem_Eq_6 fires. Mem_Eq_1 means that there is an atom $P(X, Y)$ with $X = Y$, and Mem_Eq_6 means that there is an atom $Q(X, Y)$ with $X = Y$. Therefore, the extracted rule would be $P(X, X) \vee Q(Y, Y) \Rightarrow R$.

It is important to note that, by opposition to common symbolic machine learning techniques. ANNs learn all the possibles rules in the same time. For example, in this previous example both of the rules $P(X, X) \Rightarrow R$ and $Q(X, X) \Rightarrow R$ explains the example, but the systems actually learn the complete fusion of these rules : $P(X, X) \vee Q(Y, Y) \Rightarrow R$.

8.4 Other kind of artificial neural network

All the techniques developed are using attractor neurons. The use of other kind of neuron, and/or the combination may provide solutions for more powerful rules and/or more compact artificial neural networks.

8.5 Improvement of the loading of atoms

To load a set of input atoms in the system, several combination of the input atoms have to be presented sequentially to the input layer of the ANN. Depending on the redundancy of atom with the same predicate, the number of combination can be relatively important.

Let's consider the worts case. Suppose a set I of n input atoms with the same predicate P, and suppose that the system allows m replications of atom with the same predicate in the body of the rule to infer (m is a constant for a given artificial neural network). To load I in the system, n^m combinations of atoms will have

to be presented.

Therefore, a possible improvement of this work is the following one. Since the loading of the atoms have to be sequential, sequential optimization can be used to improve it.

Chapter 9

Bibliography

[1] A. S. Davila Garcez A, K. Broda A, and D. M. Gabbay B. Symbolic knowledge extraction from trained neural networks: A sound approach.

[2] J. Barwise, editor. *Handbook of Mathematical Logic*. 1977.

[3] Artur S. d'Avila Garcez, Luís C. Lamb, and Dov M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Cognitive Technologies. Springer, 2009.

[4] Artur S. d'Avila Garcez, Lus C. Lamb, and Dov M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Springer Publishing Company, Incorporated, 2008.

[5] Pascal Hitzler, Steffen Hlldobler, and Anthony Karel Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2:2004, 2004.

[6] Katsumi Inoue. Induction as consequence finding. *Machine Learning*, 55(2):109–135, 2004.

[7] Jens Lehmann, Sebastian Bader, and Pascal Hitzler. Extracting reduced logic programs from artificial neural networks. In *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy 05*, 2005.

[8] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[9] S.H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[10] Steffen H Olldobler, Yvonne Kalinke, and Wissensverarbeitung Ki Informatik. Towards a new massively parallel computational model for logic programming.

[11] Geoffrey G. Towell and Jude W. Shavlik. Running head: Knowledge-based artificial neural networks.