4. SYSTEME INSS

Dans le chapitre précédent, nous avons présenté différents systèmes hybrides neuro-symboliques, suivis d'une discussion à propos des fonctionnalités proposées dans deux systèmes en particulier : SYNHESYS et KBANN. Nous avons essayé de tirer profit des aspects intéressants présentés par ces deux systèmes, de façon à créer un nouveau SHNS plus performant. Ce nouveau système que nous avons développé a été nommé ***INSS*** pour ***Incremental Neuro-Symbolic System***.

Le système INSS est un système hybride neuro-symbolique qui utilise un mode d'intégration par co-traitement et qui possède des modules capables de réaliser le transfert de connaissances entre les deux modules symbolique et connexionniste (voir Figure 32). Nous l'avons conçu sous l'environnement UNIX (Sun Solaris), et plus récemment il a été aussi porté sur des machines du type IBM-PC. Il a été développé initialement au sein du laboratoire LIFIA (Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle de l'IMAG/INPG, qui a été démembré en 1995 donnant l'origine à d'autres laboratoires) et actuellement il reste toujours en développement au sein de l'équipe Réseaux d'Automates du laboratoire LEIBNIZ. Le système INSS est un outil pour la construction de systèmes experts. Ses principales applications sont dans le domaine des problèmes de classification, de l'aide au diagnostic, ou de l'aide à la décision. Cependant, il a été aussi employé dans des tâches diverses, telles que le contrôle de robots autonomes et l'étude de modèles cognitifs du comportement humain (voir Chapitre 5).

MS $\Rightarrow$ transfert $\Rightarrow$ MC (Transformation de Règles en Réseau)
MS $\Leftarrow$ transfert $\Leftarrow$ MC (Explicitation de Règles du Réseau)

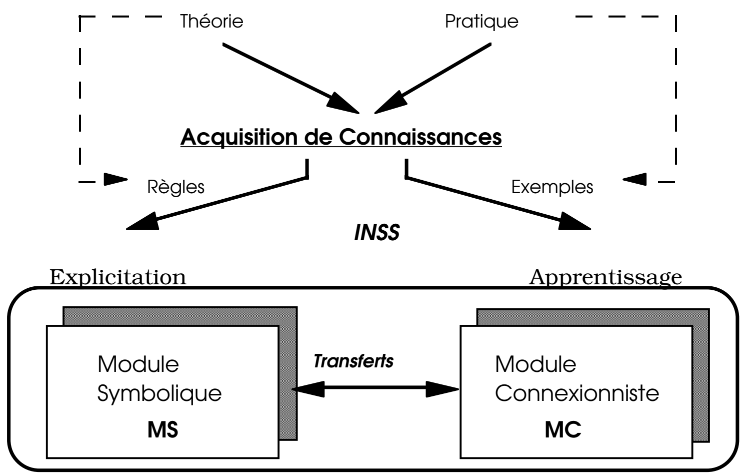

Figure 32 - Schéma Global du Système Hybride INSS

Le système INSS a été inspiré par le modèle développé par G. Towell pour les systèmes KBANN [TOW 91]. Il essaie d'améliorer leurs points faibles en ajoutant de nouvelles propriétés. Au contraire du système KBANN qui utilise l'algorithme de la Rétro-Propagation [RUM 86b], fondé sur des réseaux statiques, INSS utilise la méthode d'apprentissage Cascade-Correlation (CasCor) [FAH 90]. Cet autre algorithme permet l'ajout de nouvelles unités (neurones) au réseau pendant l'apprentissage - c'est une méthode constructive. Notre approche nous a permis d'obtenir un réseau capable de faire évoluer sa structure, ainsi que ses connaissances, tout en restant parfaitement intégré dans le cadre d'un système hybride neuro-symbolique. Par conséquent, le système INSS a l'avantage de permettre un apprentissage incrémental et aussi une extraction incrémentale de règles. Les principales améliorations de notre système par rapport à ses prédécesseurs, sont : l'utilisation de l'algorithme Cascade-Correlation ; l'extraction incrémentale de connaissances ; et la validation des nouvelles connaissances acquises par le système. Tout cela nous a permis d'implémenter un *système hybride neuro-symbolique d'acquisition constructive de connaissances.* Le système INSS constitue donc une nouvelle approche de système d'apprentissage automatique.

Le système INSS est un système hybride qui permet le transfert de *"connaissances théoriques"*, représentées par un ensemble de règles symboliques, d'un module symbolique (MS) à un module connexionniste (MC). Pour réaliser ce transfert on utilise un convertisseur capable de transformer des règles symboliques en réseau de neurones (ensemble d'unités liées par des connexions pondérées). Le réseau obtenu de cette façon pourra subir un apprentissage supervisé à partir d'un ensemble d'exemples, qui constitue ce qu'on appelle les *"connaissances pratiques"*. Ainsi, on va pouvoir réaliser un raffinement des connaissances théoriques antérieurement acquises. Après l'amélioration des connaissances du réseau par apprentissage de la base de connaissances pratiques, on peut appliquer à ce réseau un algorithme d'explicitation de règles. Cet algorithme va extraire les règles qui représentent les nouvelles connaissances acquises par le réseau. Ces règles peuvent être réinsérées dans le module symbolique, bouclant ainsi la boucle.

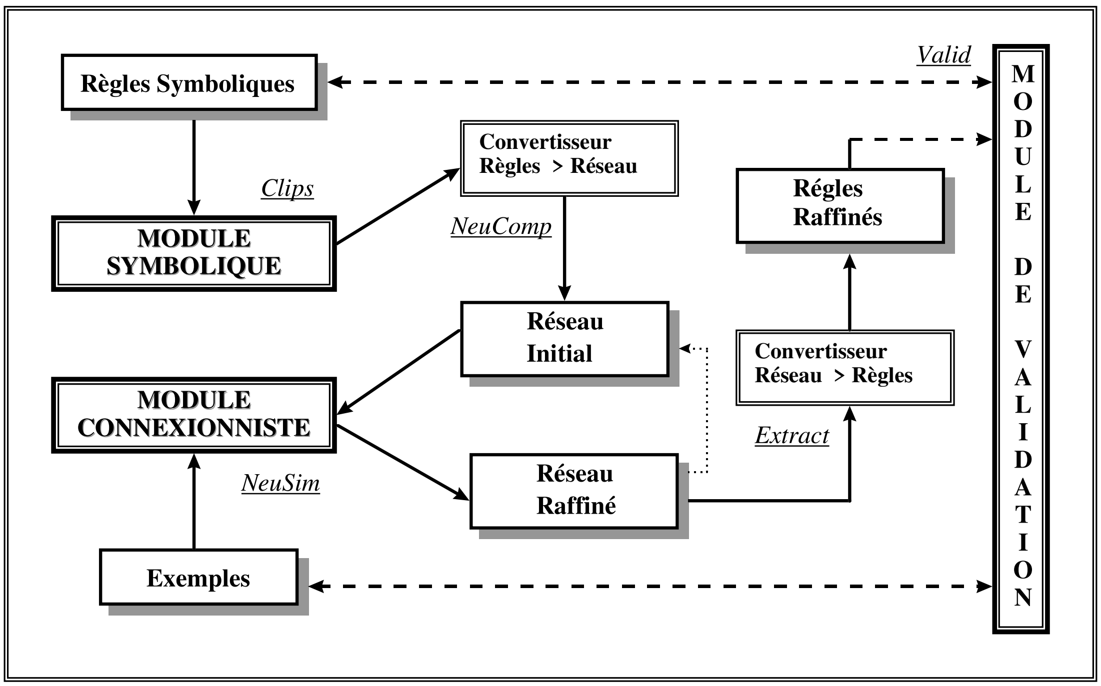

Figure 33 - L'Acquisition Constructive de Connaissances dans le Système INSS

Le système INSS est composé de deux modules principaux, le Module Symbolique et le Module Connexionniste, et en plus des modules de transfert et de validation de connaissances. La Figure 33 et la Figure 34 présentent un schéma des composants du système INSS. Le Module Symbolique est le seul module d'INSS qui n'a pas été développé par nous (voir Section 4.1). Toutes les autres composantes du système ont été réalisées par nous, à savoir :

- *NeuComp* : il réalise l'insertion (traduction / compilation) de règles dans un réseau connexionniste;

• *NeuSim* : c'est le Module Connexionniste proprement dit du système INSS. Il est responsable de la simulation et de l'apprentissage des réseaux;

• *Extract* : il permet d'extraire de règles à partir des réseaux connexionnistes,

• *Valid* : il fait la validation des nouvelles connaissances acquises par rapport aux anciennes. Il utilise tout l'ensemble de connaissances disponible sur le problème (règles ⇔ exemples).

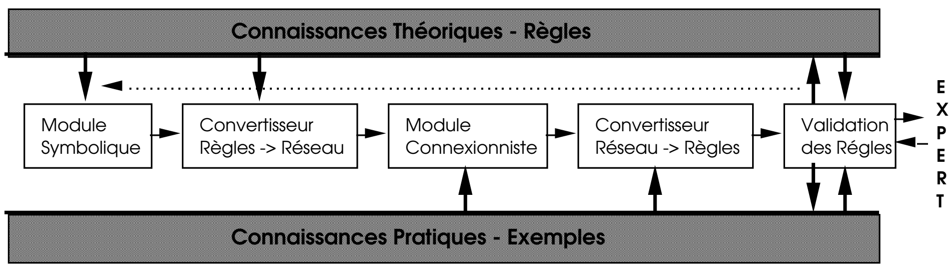

Figure 34 - Les Connaissances dans le Système INSS

Il est important de bien voir que le système comporte deux boucles principales de fonctionnement. Une première, la "petite boucle", correspond à l'entraînement du réseau connexionniste. Comme nous le verrons par la suite, dans cette boucle les nouvelles connaissances présentées au réseau sont gelées au fur et à mesure qu'elles sont apprises. La deuxième boucle, la "grande boucle", correspond, elle, au parcours complet de la chaîne de traitement des connaissances dans le système : compilation, apprentissage, extraction, validation. Dans cette deuxième boucle, les connaissances ne sont pas forcément gelées. Au terme de la validation, on peut arriver à la conclusion que de nouvelles règles doivent remplacer des règles initiales qui s'avèrent non valides. Dans ce cas il faut de nouveau compiler la base de règle modifiée en conséquence. On obtient alors un nouveau réseau qu'il faut à son tour soumettre à l'apprentissage.

Chacun des composants du système INSS va être présenté plus en détails dans les sections suivantes de ce chapitre.

4.1 MODULE SYMBOLIQUE

Le moteur d'inférence symbolique utilisé dans INSS est basé sur le système CLIPS (*C Language Integrated Production System*) [GIA 93,CUL 93]. CLIPS est un langage outil pour la création de systèmes experts qui a été développé au *Lyndon B. Jonhson Space Center* de la *NASA,* au sein de la STB (*Software Technology Branch, ex- Artificial Intelligence Section*), depuis 1984. Nous avons utilisé la version 6.0 de CLIPS disponible pour l'environnement Unix, mais qui est aussi disponible en versions adaptées aux machines du type IBM-PC ou Macintosh.

Comme son nom indique, CLIPS a été écrit dans le 'Langage C', ceci entraînant une grande portabilité et une bonne vitesse d'exécution. Ses qualités et fonctionnalités font qu'il est utilisé dans plusieurs sites de la NASA, dans de nombreux bureaux militaires et fédéraux aux Etats-Unis, ainsi que dans certaines universités et entreprises. Ce système est distribué librement aux universités et aux centres de recherche, et on peut même avoir accès au code source de tout le logiciel. Le système CLIPS est à l'origine d'autres produits dérivés, à savoir : le *Fuzzy-CLIPS* [IIT 96] développé au IIT (Institute for Information Technology) - CNRC, Canada ; le *JESS* (Java Expert System Shell) [SAN 97,WAT 97] développé au Sandia National Laboratories de Livermore aux Etats-Unis ; et l'*ECLIPSE* [HAL 97] développé par The Halley Enterprise aux Etats-Unis.

Le système CLIPS présente l'avantage de combiner trois méthodologies différentes de représentation de connaissances et de manipulation des informations. Originellement, il permettait seulement l'utilisation de règles symboliques avec un moteur d'inférence par chaînage avant basé sur l'algorithme Rete [FOR 82]. Il a ensuite acquis dans ses versions ultérieures les possibilités de programmation procédurale (comme en C ou en Lisp). Puis finalement la représentation et la manipulation de connaissances utilisant le concept d'objets ont été intégrées au système. Donc CLIPS est un outil très puissant qui permet la représentation de connaissances par des règles et par des objets, ainsi que la manipulation de ces informations à travers l'implémentation d'un mécanisme d'inférence par chaînage avant ou à travers de la programmation procédurale.

```
(clear)
(reset)
(set-strategy depth)

(assert (cas 1 FALSE FALSE Out_False))
(assert (cas 2 TRUE  FALSE Out_True))
(assert (cas 3 FALSE TRUE  Out_True))
(assert (cas 4 TRUE  TRUE  Out_False))

(defrule xor
   ?nfact <- (cas ?x ?x1 ?x2 ?Out)
   (test (and (or ?x1 ?x2) (not (and ?x1 ?x2))) )
  =>
   (printout t "Cas: " ?x " # Clips: Out_True # Exemple: " ?Out crlf)
   (retract ?nfact) )

(defrule not-xor
   ?nfact <- (cas ?x ?x1 ?x2 ?Out)
   (test (not (and (or ?x1 ?x2) (not (and ?x1 ?x2)))) )
  =>
   (printout t "Cas: " ?x " # Clips: Out_False # Exemple: " ?Out crlf)
   (retract ?nfact) )

(facts)
(run)
```

Figure 35 - Exemple de programme en CLIPS (Fonction Xor)

Dans le cadre du système INSS, nous n'avons utilisé qu'une partie des potentialités de CLIPS. Ce système a servi uniquement comme moteur d'inférence de notre SHNS et dans la réalisation du processus de validation de connaissances. La syntaxe utilisée par CLIPS pour décrire les règles et les faits (les exemples) est un peu différente de la syntaxe que nous avons utilisée dans les outils que nous avons développés nous-mêmes. La conversion d'un format à l'autre de description des règles et des exemples peut être faite facilement à travers l'utilisation de convertisseurs des fichiers utilisés par CLIPS et par INSS. La Figure 35 montre un exemple de fichier CLIPS décrivant la base de connaissances associée au problème du XOR (Ou Exclusif). Dans l'Annexe A, nous présentons des exemples complets des fichiers CLIPS et INSS appliqués à ce même problème.

Nous allons centrer nos discussions plus sur les autres modules d'INSS (NeuComp, NeuSim, Extract et Valid) que sur le Module Symbolique, car ce dernier module est un système bien développé et étudié, et l'on peut trouver une vaste documentation sur le sujet. De plus, le module NeuSim du système INSS peut être utilisé à la place de CLIPS, car une fois que les règles sont insérées dans le Module Connexionniste, celui-ci peut être utilisé comme un moteur d'inférence symbolique.

4.2 INSERTION DE CONNAISSANCES A PRIORI

Le module *NeuComp* (Neural Compiler) permet d'obtenir un RNA à partir d'un ensemble de règles symboliques. La transformation de règles en réseau suit les mêmes principes que dans le système KBANN (voir Section 3.3.2.1). On obtient un réseau où sont spécifiés les connexions et ses poids respectifs. Ce réseau est construit de façon à obtenir par son activation un comportement qui correspond exactement aux résultats qu'on peut obtenir à partir de l'utilisation de la base de règles d'origine. Nous avons utilisé une syntaxe très proche de celle employée dans la représentation des règles symboliques de KBANN, comme on peut le constater dans l'exemple présenté ci-dessous.

```
% NeuComp File: Xor.Symb

$Features:
 X1:binary;
 X2:binary.
$End_Feat.

$Rules:
 X1_OR_X2   <- X1;
 X1_OR_X2   <- X2;
 X1_AND_X2  <- X1,X2;
 XOR        <- X1_OR_X2,Not(X1_AND_X2).
$End_rules.

$End.
```

Figure 36 - Règles symboliques utilisées par NeuComp

L'exemple ci-dessus montre des propositions d'ordre 0 décrites selon la syntaxe acceptée par NeuComp. Les règles qui ont les mêmes conséquents représentent des disjonctions ('ou' logique) entre ses antécédents, et les conjonctions ('et' logique) sont représentées par des antécédents séparés par des virgules. Donc, les règles représentées dans l'exemple présenté ci-dessus peuvent être interprétées de la façon suivante :

Si (X1 ou X2) et non (X1 et X2) Alors XOR est Vrai.

Nous avons étendu le type de règles symboliques accepté par NeuComp de façon à pouvoir travailler avec des règles de "plus haut niveau", c'est-à-dire, avec des règles d'ordre 0^+ (inclusion de la notion d'intervalle et utilisation de prémisses formées par des valeurs continues). Ainsi, Neucomp accepte aussi des règles de production avec des comparaisons du type suivant :

<Attribut> <opérateur> <Valeur> ou *<Attribut> <opérateur> <Attribut>*

(en utilisant des opérateurs de comparaison tels que : Plus-Grand-Que, Plus-Petit-Que, Egal)

Nous allons décrire les processus de conversion de ce type de règles vers des réseaux de type PMC dans la Section 4.2.1. Afin de donner une meilleure description des règles symboliques que nous utilisons avec NeuComp, la Figure 37 montre un résumé de la syntaxe et des différents type de règles acceptées tandis que la Figure 38 montre un exemple pratique plus complet d'un fichier de règles symbolique d'INSS.

```
  Conséquent <- Antécédent [ , Antécédent, ... ] ;
  Conséquent <= Antécédent [ , Antécédent, ... ] ;
  Conséquent <- Fonction ( Antécédent ), Fonction ( Antécédent, Liste-de-Paramètres );
  Conséquent <- Antécédent (True), Antécédent (False);

où,

  '<-' : Représente une règle adaptable (peut être modifiée au cours de l'apprentissage)
  '<=' : Représente une règle statique (n'est pas modifiée au cours de l'apprentissage)
  Fonction : Choisie parmi un des opérateurs suivants
             Equal, In, In_Range, Ntrue, NofM, MofN, Not
             LT, Less_Than, LTOE, Less_Or_Equal,
             GT, Greater_Than, GTOE, Greater_Or_Equal
```

Figure 37 - Aperçu de la syntaxe utilisée par Neucomp

La Figure 38 montre aussi que NeuComp accepte plusieurs types de données différents comme entrées des réseaux : type 'binary' (valeurs booléennes), type 'nominal' (valeurs discrètes), type 'range' (valeurs continues), type 'continuous' (série ordonnée de valeurs continues) et type 'ordered' (série ordonnée de valeurs discrètes). Les variables du type 'binary' et 'range' sont codées dans une seule unité d'entrée, tandis que les variables du type 'nominal' nécessitent une unité d'entrée pour coder chaque valeur nominale (e.g. une variable d'entrée couleur, pouvant assumer trois valeurs, rouge, verte ou bleue, va être codée par trois unités d'entrée : couleur#rouge, couleur#verte, couleur#bleue ; seulement une des ces trois entrées va pouvoir être activée à la fois). Les variables du type 'ordered' et 'continuous' n'ont pas été beaucoup étudiées par nous jusqu'à présent, car elles peuvent aussi être représentées en utilisant respectivement les types 'nominal' et 'range' (ce dernier avec l'utilisation de la fonction In_Range).

```
% NeuComp File: Example.Symb

#define MINVALUE -40.0
#define MAXVALUE  40.0

$Features:
  respiration : binary;
  heart       : nominal    # 2 : beat, stopped;
  temperature : range          : [0,45];
  extern_temp : range          : [MINVALUE, MAXVALUE];
  pulse       : continuous # 3 : low[0,30], medium[30,60], high[60,90];
  day         : ordered    # 7 : mon, tue, wed, thu, fri, sat, sun.
$End_Feat.

$Rules:
  dead  <= NOT (heart(beat));
  alive <= heart(beat), respiration(yes);
  dead  <- respiration(false);
  dead  <- EQUAL(temperature, extern_temp);
  alive <- heart(beat), NOT(dead);
  alive <- IN_RANGE(temperature, 35, 40), pulse(medium);
  alive <- NTRUE(2, heart(beat), pulse(medium), respiration(yes));
  alive <- GTOE(temperature, 35);
  cold  <- Less_Than(extern_temp, 0.0);
  hot   <- GT(extern_temp, 25.0);
  weekend <- day(sat);
  weekend <- day(sun);
  have_a_good_time <- weekend, hot, alive.

#include "other-rules.symb"

$End_rules.
$End.
```

Figure 38 - Exemple de fichier de règles symboliques (structures syntaxiques)

Le module NeuComp utilise plusieurs fichiers : des fichiers d'entrée (*ccfg* : spécifie les paramètres de la compilation ; *symb* : base de règles symboliques) et des fichiers de sortie (*wts* : poids des connexions des unités du réseau compilé ; *top* : topologie du réseau et symboles associés à chaque unité ; *rep* : rapport d'accompagnement du processus de compilation ; *gnup* : fichier graphique compatible avec 'gnuplot' [LIA 94, KON 96], il affiche la topologie du réseau obtenu ; *tmp* : fichier temporaire crée par le pré-processeur). Le fichier des paramètres de compilation nous permet de spécifier les options de compilation, selon l'exemple présenté ci-dessous :

```
%
% >> Neucomp : Options de Compilation <<
% File: Exemple.ccfg
%
$Flags [DGPO]
$WF    4.0
%
% Flags:
%    D - Afficher les tables de symboles créées pendant la compilation
%    G - Créer un fichier graphique avec la topologie du réseau (compatible avec gnuplot)
%    P - Utiliser le pré-processeur du langage C (cpp - directives: #include et #define)
%    C - Connecter toutes les entrées et/ou les unités du réseau entre deux couches
%    I  - Créer seulement la topologie du réseau et initialiser les poids aléatoirement
%    O - Créer les entrées dans le même ordre utilisé pour les déclarer dans 'features'
%
% WF:
%    Valeur du "Weight_Factor" (valeur constante 'W' utilisée dans KBANN)
%
```

Figure 39 - Fichier de configuration du compilateur NeuComp

Les fichiers décrivant des poids (*.wts) et la topologie du réseau (*.top) vont ensuite pouvoir être lus par NeuSim. Ils établissent ainsi une configuration initiale du réseau avant de commencer l'apprentissage à partir de la base d'exemples. Dans l'Annexe B, nous présentons des exemples de chacun de ces fichiers.

4.2.1 Compilation de Règles d'Ordre 0^+

Une des principales contributions de nos recherches par rapport au modèle proposé par Towell [TOW 91], est l'extension de la capacité de traitement des bases de règles utilisées par KBANN. Nous avons développé une méthode qui permet d'insérer des règles d'ordre 0^+ dans nos réseaux, le tout d'une façon compatible avec le mode de fonctionnement des réseaux du type KBANN. Nous proposons des nouvelles méthodes de compilation de règles en réseaux, afin d'ajouter les fonctions suivantes à nos bases de règles :

1. **Greater_Than (Trait, ValCte, [Paramètres])** : l'unité qui représente cette fonction va s'activer si la valeur de la variable d'entrée *'Trait'* est supérieur à la valeur constante *'ValCte'* spécifiée comme paramètre de cette fonction. Une variante de cette fonction est la fonction

Greater_Or_Equal, dans laquelle la sortie de l'unité est activée seulement si la variable d'entrée *'Trait'* est égale ou supérieur à la valeur constante *'ValCte'*. Les deux fonctions décrites ci-dessus possèdent des noms abrégés, reconnus par le compilateur : GT et GTOE. Cette unité du type Greater_Than possède une seule entrée et une sortie, et la valeur de son poids ainsi que la valeur de son seuil sont obtenues de la façon suivante :

Poids_GT(T,V) = Sensibilité

Seuil_GT(T,V) = -1.0 * (ValCte - Confiance) * Sensibilité

Sortie de l'unité: S = Fct_Sigmoïde (Trait * Poids_GT + Seuil_GT)

Les valeurs de la *'sensibilité'* et de la *'confiance'* sont des paramètres optionnels définis dans la fonction en question. Nous allons décrire ces paramètres et son influence par rapport au comportement des fonctions, à la fin de cette section.

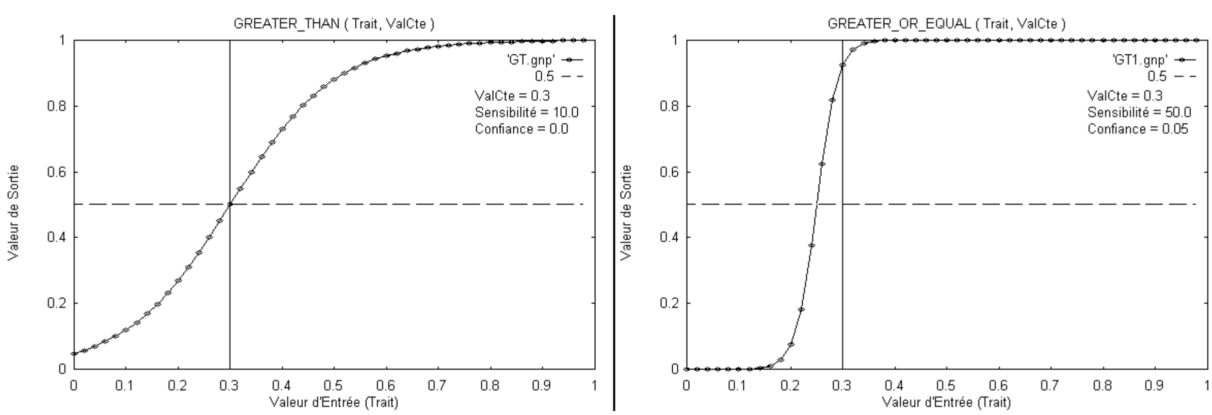

Figure 40 - Fonction Greater_Than (Trait, Valeur)

2. **Greater_Than (Trait1, Trait2, [Paramètres])** : l'unité qui représente cette fonction va s'activer si la valeur de la variable d'entrée *'Trait1'* est supérieure à la valeur de la variable d'entrée *'Trait2'*. Une variante de cette fonction est la fonction *Greater_Or_Equal*, dans laquelle la sortie de l'unité est activée seulement si la variable d'entrée *'Trait1'* est égale ou supérieur à la valeur de la variable d'entrée *'Trait2'*. Les deux fonctions décrites ci-dessus possèdent des noms abrégés, reconnus par le compilateur : GT et GTOE. La seule différence entre cette fonction et la

fonction précédente est au niveau des paramètres, tels que le compilateur va automatiquement déterminer le type de fonction à adopter. Cette unité du type Greater_Than possède deux entrées et une seule sortie, et la valeur de ses poids ainsi que la valeur de son seuil sont obtenus de la façon suivante :

Poids1_GT(T,T) = Sensibilité

Poids2_GT(T,T) = -1.0 * Sensibilité

Seuil_GT(T,T) = Confiance * Sensibilité

Sortie: S = Fct_Sigmoïde(Trait1*Poids1_GT + Trait2*Poids2_GT + Seuil_GT)

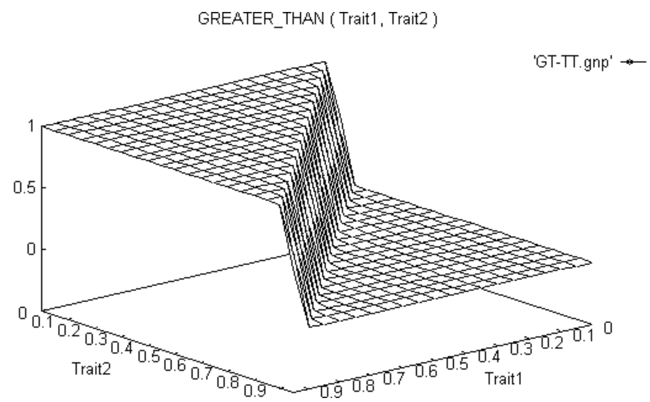

Figure 41 - Fonction Greater_Than (Trait, Trait)

3. **Less_Than (Trait, ValCte, [Paramètres])** : l'unité qui représente cette fonction va s'activer si la valeur de la variable d'entrée *'Trait'* est inférieur à la valeur constante *'ValCte'* spécifiée comme paramètre de cette fonction. Une variante de cette fonction est la fonction *Less_Or_Equal*, dans laquelle la sortie de l'unité est activée seulement si la variable d'entrée *'Trait'* est inférieure ou égale à la valeur constante *'ValCte'*. Les deux fonctions décrites ci-dessus possèdent des noms abrégés, reconnus par le compilateur : LT et LTOE. Cette unité du type Less_Than possède une seule entrée et une sortie, et la valeur de son poids ainsi que la valeur de son seuil sont obtenus de la façon suivante :

Poids_LT(T,V) = -1.0 * Sensibilité

Seuil_LT(T,V) = (ValCte + Confiance) * Sensibilité

Sortie: S = Fct_Sigmoïde (Trait * Poids_LT + Seuil_LT)

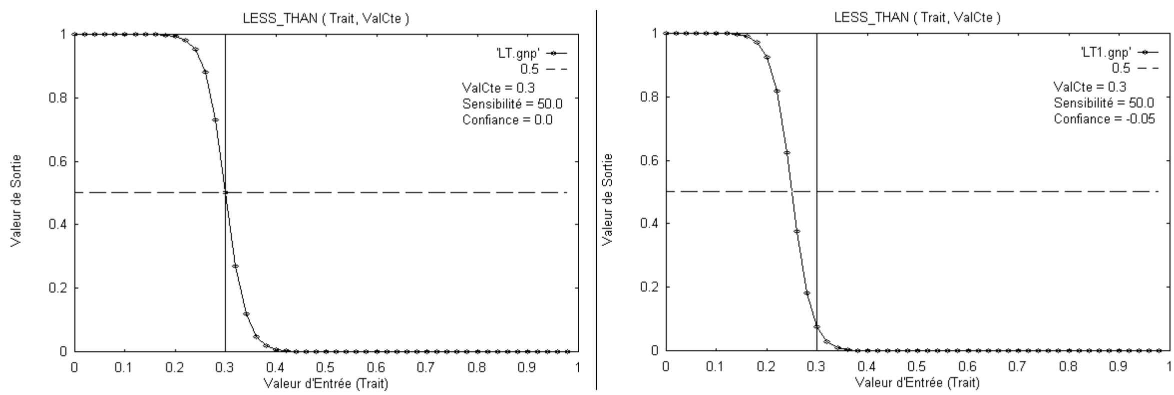

Figure 42 - Fonction Less_Than (Trait, Valeur)

4. **Less_Than (Trait1, Trait2, [Paramètres])** : l'unité qui représente cette fonction va s'activer si la valeur de la variable d'entrée *'Trait1'* est inférieure à la valeur de la variable d'entrée *'Trait2'*. Une variante de cette fonction est la fonction *Less_Or_Equal*, dans laquelle la sortie de l'unité est activée seulement si la variable d'entrée *'Trait1'* est inférieur ou égale à la valeur de la variable d'entrée *'Trait2'*. Les deux fonctions décrites ci-dessus possèdent des noms abrégés, reconnus par le compilateur : LT et LTOE. La seule différence entre cette fonction et la fonction précédente est au niveau des paramètres : le compilateur va automatiquement déterminer le type de fonction à adopter. Cette unité du type Less_Than possède deux entrées et une seule sortie, et la valeur de ses poids ainsi que la valeur de son seuil sont obtenues de la façon suivante :

Poids1_LT(T,T) = -1.0 * Sensibilité

Poids2_LT(T,T) = Sensibilité

Seuil_LT(T,T) = Confiance * Sensibilité

Sortie: S = Fct_Sigmoïde(Trait1*Poids1_LT + Trait2*Poids2_LT + Seuil_LT)

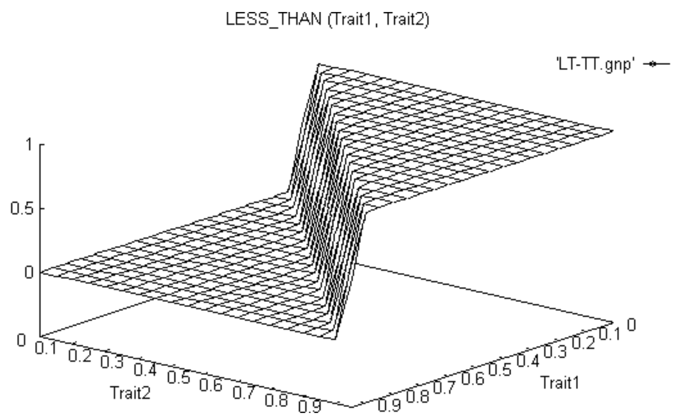

Figure 43 - Fonction Less_Than (Trait, Trait)

5. **In_Range (Trait, ValCte1, ValCte2, [Paramètres]**) : l'unité qui représente cette fonction va s'activer si la valeur de la variable d'entrée *'Trait'* est comprise dans l'intervalle spécifié par les valeurs constantes *'ValCte1'* et *'ValCte2'*. La fonction In_Range est réalisé en utilisant la combinaison (et logique) de deux unités *: "Si GT (Trait, ValCte1) et LT (Trait, ValCte2) Alors In_Range (Trait, ValCte1, ValCte2) est vrai"*. Cette fonction possède aussi un nom abrégé, reconnu par le compilateur : IN. Les trois unités qui composent la fonction In_Range sont définies de la façon suivante :

Unité GT (Trait, ValCte1):

Poids_GT(T,V) = Sensibilité

Seuil_GT(T,V) = -1.0 * (ValCte1 - Confiance) * Sensibilité

Sortie GT (Trait, ValCte1) = Fct_Sigmoïde (Trait * Poids_GT + Seuil_GT)

Unité LT (Trait, ValCte2):

Poids_LT(T,V) = -1.0 * Sensibilité

Seuil_LT(T,V) = (ValCte2 + Confiance) * Sensibilité

Sortie LT (Trait, ValCte2) = Fct_Sigmoïde (Trait * Poids_LT + Seuil_LT)

<u>Unité Et (GT, LT)</u>:

Poids1_GT(T,V) = WF

Poids2_LT(T,V) = WF

Seuil_GT(T,V) = -1.5 * WF

Sortie Et (GT,LT) = Fct_Sigmoïde (GT(Trait, ValCte1) * Poids1_GT +
LT(Trait, ValCte2) * Poids2_LT + Seuil_GT)

La valeur *'WF'* , utilisée dans une des unités qui représente cette fonction, est une valeur constante spécifiée dans le fichier de configuration du compilateur (*weight_factor*). La valeur du paramètre 'confiance' détermine si l'intervalle considéré doit inclure ou pas les valeurs des extrémités (ValCte1 et ValCte2).

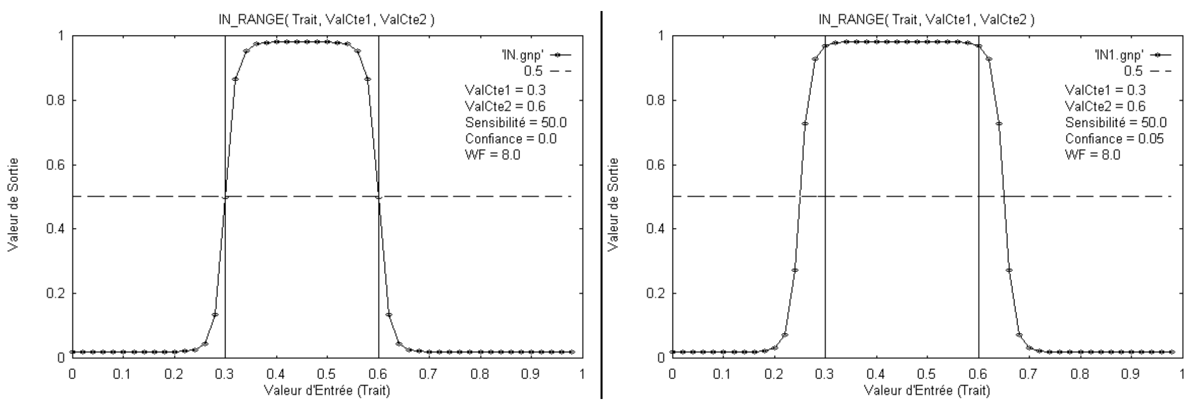

Figure 44 - Fonction Equal (Trait, Valeur)

6. **Equal (Trait, ValCte, [Paramètres])** : l'unité qui représente cette fonction va s'activer si la valeur de la variable d'entrée *'Trait'* est égale à la valeur constante spécifiée par *'ValCte'* . Nous considérons que deux valeurs sont égales s'ils sont "suffisamment proches". La fonction Equal est réalisé en utilisant la combinaison ('et' logique) de deux unités :

"Si GT (Trait, ValCte) et LT (Trait, ValCte) Alors Equal (Trait, ValCte) est vrai"

ou bien,

"Si In_Range(Trait, ValCte, ValCte) Alors Equal (Trait, ValCte)"

Dans ce cas de figure, les fonctions LT et GT doivent obligatoirement inclure la valeur de l'extrémité (ValCte : ValCte1 = ValCte2), sinon la fonction va retourner toujours une valeur de sortie 'fausse' (inactive). Le paramètre 'confiance' détermine le dégréé de proximité qu'on doit avoir par rapport à la valeur constante en question. Les trois unités qui composent la fonction Equal sont définies de la façon suivante :

Unité GT (Trait, ValCte):

Poids_GT(T,V) = Sensibilité

Seuil_GT(T,V) = -1.0 * (ValCte - Confiance) * Sensibilité

Sortie GT (Trait, ValCte) = Fct_Sigmoïde (Trait * Poids_GT + Seuil_GT)

Unité LT (Trait, ValCte):

Poids_LT(T,V) = -1.0 * Sensibilité

Seuil_LT(T,V) = (ValCte + Confiance) * Sensibilité

Sortie LT (Trait, ValCte) = Fct_Sigmoïde (Trait * Poids_LT + Seuil_LT)

Unité Et (GT, LT):

Poids1_GT(T,V) = WF

Poids2_LT(T,V) = WF

Seuil_GT(T,V) = -1.5 * WF

Sortie Et (GT,LT) = Fct_Sigmoïde (GT(Trait, ValCte) * Poids1_GT +
LT(Trait, ValCte) * Poids2_LT + Seuil_GT)

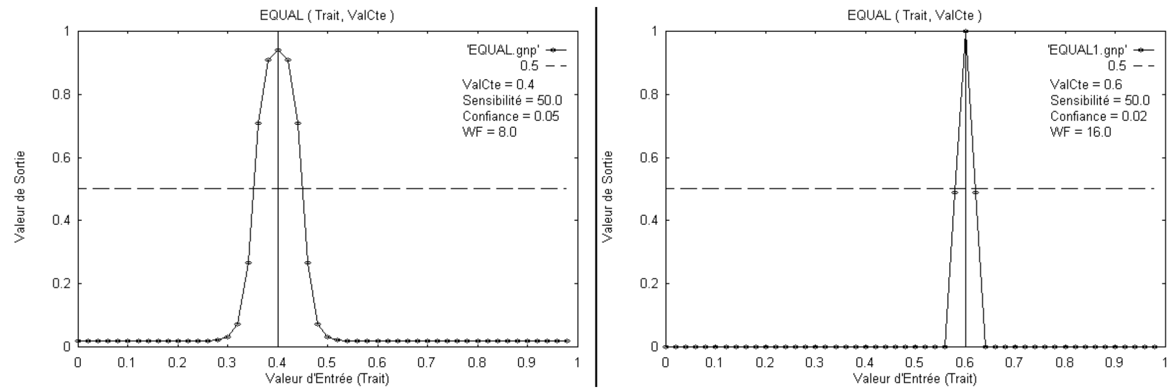

Figure 45 - Fonction Equal (Trait, Valeur)

7. **Equal (Trait1, Trait2, [Paramètres])** : l'unité qui représente cette fonction va s'activer si la valeur de la variable d'entrée *'Trait1'* est égale à la valeur de la variable d'entrée *'Trait2'*. Nous considérons que deux valeurs sont égales s'ils sont "suffisamment proches". La seule différence entre cette fonction et la fonction précédente est au niveau des paramètres, où le compilateur va automatiquement déterminer le type de fonction à adopter. La fonction Equal est réalisée en utilisant la combinaison ('et' logique) de deux unités : *"Si GT (Trait1, Trait2) et LT (Trait1, Trait2) Alors Equal (Trait1, Trait2) est vrai"*. Dans ce cas de figure, les fonctions LT et GT doivent obligatoirement accepter une certaine marge d'erreur (proximité), sinon la fonction va retourner toujours une valeur de sortie 'fausse' (inactive). Le paramètre 'confiance' détermine le degré de proximité que doit avoir une variable par rapport à l'autre. Les trois unités qui composent la fonction Equal sont définies de la façon suivante :

Unité GT (Trait1, Trait2):

Poids1_GT(T,T) = Sensibilité

Poids2_GT(T,T) = -1.0 * Sensibilité

Seuil_GT(T,T) = Confiance * Sensibilité

Sortie GT (Trait1, Trait2) = Fct_Sigmoïde(Trait1 * Poids1_GT + Trait2 * Poids2_GT + Seuil_GT)

Unité LT (Trait1, Trait2):

Poids1_LT(T,T) = -1.0 * Sensibilité

Poids2_LT(T,T) = Sensibilité

Seuil_LT(T,T) = Confiance * Sensibilité

Sortie LT (Trait1, Trait2) = Fct_Sigmoïde(Trait1 * Poids1_LT +
Trait2 * Poids2_LT + Seuil_LT)

Unité Et (GT, LT):

Poids1_GT(T,V) = WF

Poids2_LT(T,V) = WF

Seuil_GT(T,V) = -1.5 * WF

Sortie Et (GT,LT) = Fct_Sigmoïde (GT(Trait1, Trait2) * Poids1_GT +
LT(Trait1, Trait2) * Poids2_LT + Seuil_GT)

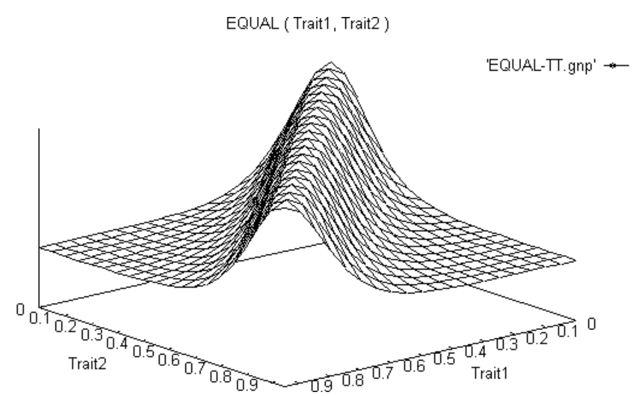

Figure 46 - Fonction Equal (Trait, Trait)

Le paramètre 'Sensibilité' permet d'ajuster la pente de la courbe obtenue à la sortie des unités qui représentent les fonctions décrites ci-dessus. La valeur de la sensibilité doit être une valeur positive égale ou supérieure à 1.0. Nous fixons cette valeur égale à 1.0 dans le cas où le paramètre sensibilité n'est pas indique dans la fonction. Une valeur de sensibilité proche de 1.0 va entraîner l'obtention d'une fonction avec une sortie quasi linéaire, tandis qu'une valeur de sensibilité plus grande (e.g. sensibilité=100.0) va entraîner l'obtention d'une fonction avec une sortie plus abrupte, qui passe d'un état à l'autre très rapidement (faux ⇒ vrai et vice versa). La Figure 47 montre un exemple des différents comportements obtenus à la sortie d'une fonction du type Greater_Than, étant donné des valeurs différentes attribuées au paramètre 'Sensibilité'.

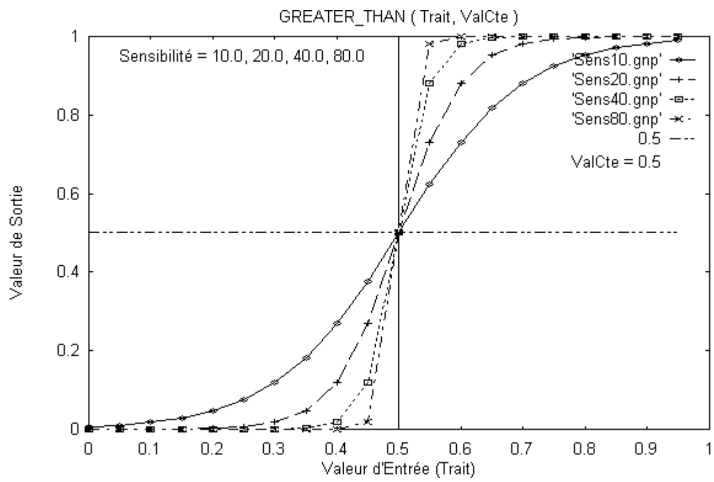

Figure 47 - Sortie de la fonction Greater_Than par rapport à la valeur de la sensibilité

Le paramètre 'Confiance' permet d'ajuster l'erreur acceptée à la sortie des unités (degré de proximité utilisé dans les comparaisons) qui représentent les fonctions décrites ci-dessus. La confiance permet de modifier la sortie des fonctions de façon à inclure ou à exclure les valeurs qui délimitent le changement d'état de la réponse. Ainsi, on peut transformer une fonction du type Greater_Than de façon à obtenir une fonction Greater_Than_Or_Equal, ou une fonction Greater_Than_And_Not_Equal, selon la valeur choisie de la confiance. La valeur de la confiance doit être une valeur positive égale ou supérieure à 0.0. Nous fixons cette valeur égale à 0.0 dans le cas où le paramètre confiance n'est pas indiqué dans la fonction. L'utilisation d'une valeur de confiance égale à 0.0 va entraîner l'obtention d'une fonction avec une valeur de sortie vraie seulement si les valeurs d'entrée satisfont parfaitement la fonction de comparaison définie ; tandis qu'une valeur de confiance plus grande va entraîner l'obtention d'une fonction avec une sortie plus souple, permettant de comparer les valeurs d'entrée avec une certaine marge d'erreur. Par exemple, si la valeur de confiance est égale à 0.1 dans une fonction du type Equal, nous allons considérer les entrées comme étant égales jusqu'à un écart maximum de 0.1 entre leurs valeurs. La Figure 48 montre un exemple des différents comportements obtenus à la sortie des fonctions du type Greater_Than et In_Range, étant donné des valeurs différentes attribuées au paramètre 'Confiance'.

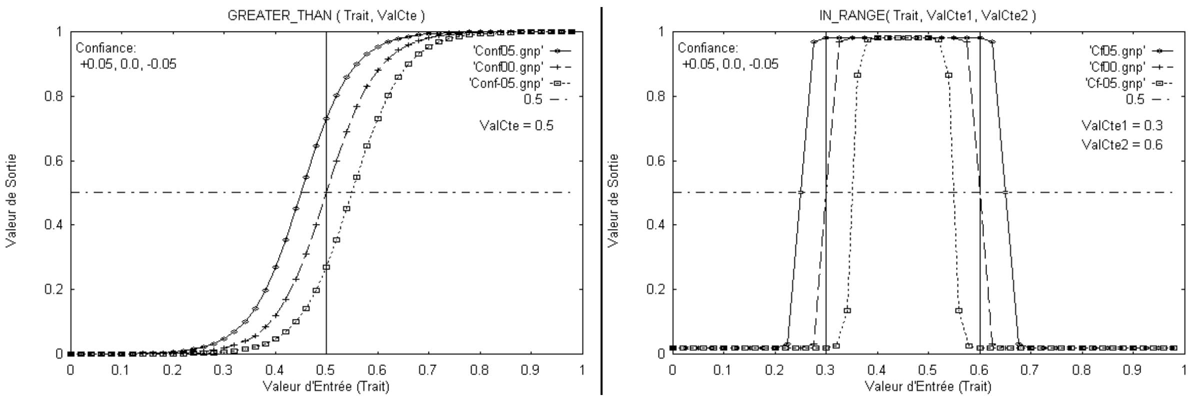

Figure 48 - Sortie des fonctions Greater_Than et In_Range par rapport à la valeur de la confiance

Le paramètre 'WF' est utilisé seulement dans les fonctions qui nécessitent la création de plus d'une unité pour obtenir le comportement de sortie de la fonction, c.-à-d. dans les fonctions du type In_Range et Equal. Ce paramètre fonctionne exactement de la même façon que la valeur constante 'W' utilisée dans les réseaux KBANN, et son rôle est d'augmenter l'écart entre les deux valeurs extrêmes de sortie de la fonction. La Figure 49 montre un exemple des différents comportements obtenus à la sortie d'une fonction du type In_Range, étant donné des valeurs différentes attribuées au paramètre 'WF'.

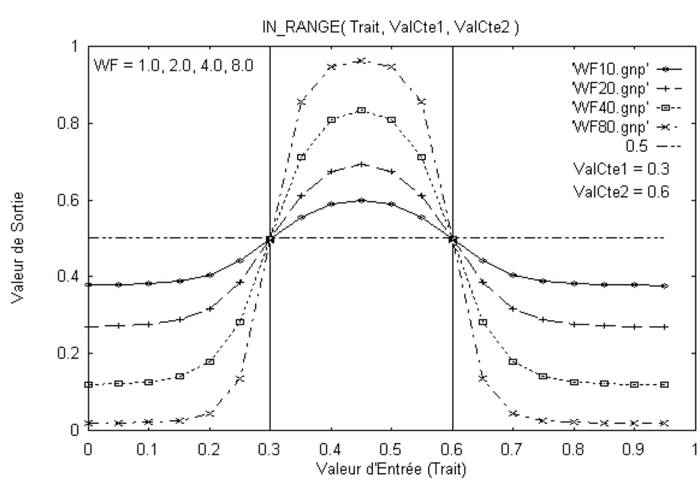

Figure 49 - Sortie de la fonction In_Range par rapport à la valeur de WF

Finalement, toutes ces fonctions que nous venons de présenter sont applicables seulement à des entrées du type 'intervalle' (Feature: range). Il est important de souligner que les paramètres 'Sensibilité', 'Confiance' et 'WF' sont aussi très liées à l'étendue de l'intervalle des valeurs qui sont fournies comme entrées. La méthode actuelle de spécification de ces paramètres n'étant pas très facile à employer, cela nous a amené à la création d'un outil (logiciel 'FnctOut') qui permet de vérifier la courbe de réponse d'une fonctions particulière, étant donné une configuration spécifique des entrées et des paramètres. Cet outil a été employé dans la création des figures présentées dans cette section. Nous envisageons de développer des méthodes capables d'estimer de façon plus automatique les valeurs à mettre dans les paramètres des fonctions.

4.2.2 Discussion sur la Compilation de Règles

Le module de compilation de règles d'INSS nous a permis d'étendre le type de règles que l'on pouvait utiliser sur les réseaux du type KBANN aux règles d'ordre 0^+. Nous avons réalisé des fonctions de conversion de règles en réseau qui nous permettent de travailler directement sur des entrées vraiment continues, ce qui n'était pas le cas des réseaux KBANN. Notre approche de la compilation de règles représente une amélioration significative des capacités de manipulation de connaissances théoriques par les réseaux connexionnistes de type PMC.

Il faut souligner que non seulement des règles d'ordre 0^+ peuvent être introduites dans nos réseaux, mais aussi des 'règles de haut niveau', c'est-à-dire des règles qui expriment des relations directes entre deux entrées. Nous ne sommes plus limités aux règles du type "*'entrée <comparaison> valeur"*, puisque nous pouvons désormais utiliser des règles du type *"entrée <comparaison> entrée"*. La compilation de ces règles d'ordre 0^+ et de haut niveau dans des réseaux peut être contrôlée par plusieurs paramètres, qui permettent le réglage fin de la fonction d'activation obtenue dans la sortie des unités. Nous avons aussi développé des outils qui permettent de visualiser les sorties des unités par rapport aux différents paramètres de compilation utilisés, ce qui va aider beaucoup l'utilisateur dans la tâche de spécification de ce type de règles.

L'extension des règles acceptées par INSS nous a permis d'appliquer notre système à des tâches où la manipulation de valeurs continues est plus importante. Tel est le cas des applications en robotique autonome (voir des exemples pratiques dans la section 5.4 et dans l'Annexe B.2). Toutefois, nous n'avons pas approfondi nos études sur la possibilité d'adapter directement les poids des règles compilées sous la forme d'un réseau, puisque nous les conservons inaltérées dans nos réseaux. Notre choix de conserver les règles inaltérées après la compilation va être discuté dans la section suivante. Donc, l'adaptation directe des règles (adaptation des poids des connexions obtenus par compilation) qui manipulent des valeurs d'entrée continues reste un sujet ouvert à l'étude.

4.3 MODULE CONNEXIONNISTE

Le module connexionniste NeuSim (Neural Simulator) permet de réaliser l'adaptation des poids d'un réseau du type PMC par apprentissage supervisé à partir d'une base d'exemples (fonctionnement en mode d'apprentissage). Ce module permet aussi d'activer un réseau afin d'obtenir la réponse produite à sa sortie, étant donné un ensemble particulier de valeurs présentées en entrée (fonctionnement en mode de consultation). Notre module connexionniste possède aussi des propriétés très intéressantes, telles que la possibilité d'adaptation de la topologie du réseau, ou l'apprentissage à l'aide d'une méthode très performante.

Le choix du modèle de réseau de neurones à utiliser au sein du système hybride INSS a été fondamental pour obtenir un module connexionniste d'acquisition constructive de connaissances. Dès lors que nous cherchions une méthode qui soit adaptée à une méthodologie constructive d'acquisition de connaissances, il nous a fallu un réseau évolutif (capable de faire évoluer son architecture ainsi que ses poids). On trouve dans la littérature plusieurs méthodes de ce type [FIE 94b, CHE 97], parmi lesquelles on peut distinguer trois classes principales:

- les algorithmes constructifs. E.g.: l'algorithme Cascade-Correlation, l'algorithme Tiling, l'algorithme Upstart et les réseaux ARN2.

- les algorithmes de simplification des réseaux. E.g.: les algorithmes d'élagage *'Optimal Brain Damage'* - OBD et *'Optimal Brain Surgeon'* - OBS.
- les algorithmes de recherche de topologie. E.g.: les algorithmes génétiques du type TopGen, Regent et Addemup [OPI 95a].

Nous avons choisi de travailler avec un algorithme du type constructif, car il semble être plus adapté à notre approche d'acquisition constructive de connaissances. Le fait d'avoir choisi ce type d'algorithme pour NeuSim, n'exclut pas la possibilité d'appliquer d'autres algorithmes dans les autres modules du système, comme par exemple pour les algorithmes de simplification qui sont utilisés dans le module d'extraction de règles à partir des réseaux.

Ainsi, nous avons choisi d'utiliser l'algorithme Cascade-Correlation (CasCor) [FAH 90] qui réalise l'adaptation des poids et de la topologie du réseau, contrairement à l'algorithme de la Rétro-Propagation, utilisé dans les réseaux KBANN, qui ne réalise qu'une adaptation des poids. Nous allons présenter l'algorithme CasCor plus en détails dans la section qui suit, et discuter les principaux avantages qu'apporte cette méthode par rapport à la méthode de la Rétro-Propagation [RUM 86].

4.3.1 Méthode d'Apprentissage Cascade-Correlation

L'algorithme CasCor [FAH 90] débute l'apprentissage avec un réseau composé uniquement par des unités d'entrée et de sorties (pas d'unités dans la couche cachée - voir Figure 50). En fait, comme on le verra plus loin, il est aussi possible de commencer l'apprentissage avec un réseau initial plus complexe. Les unités de la couche cachée sont créées petit à petit au cours de l'apprentissage. Cet algorithme réalise en alternance l'adaptation des unités de sortie et l'adaptation des unités de la couche cachée.

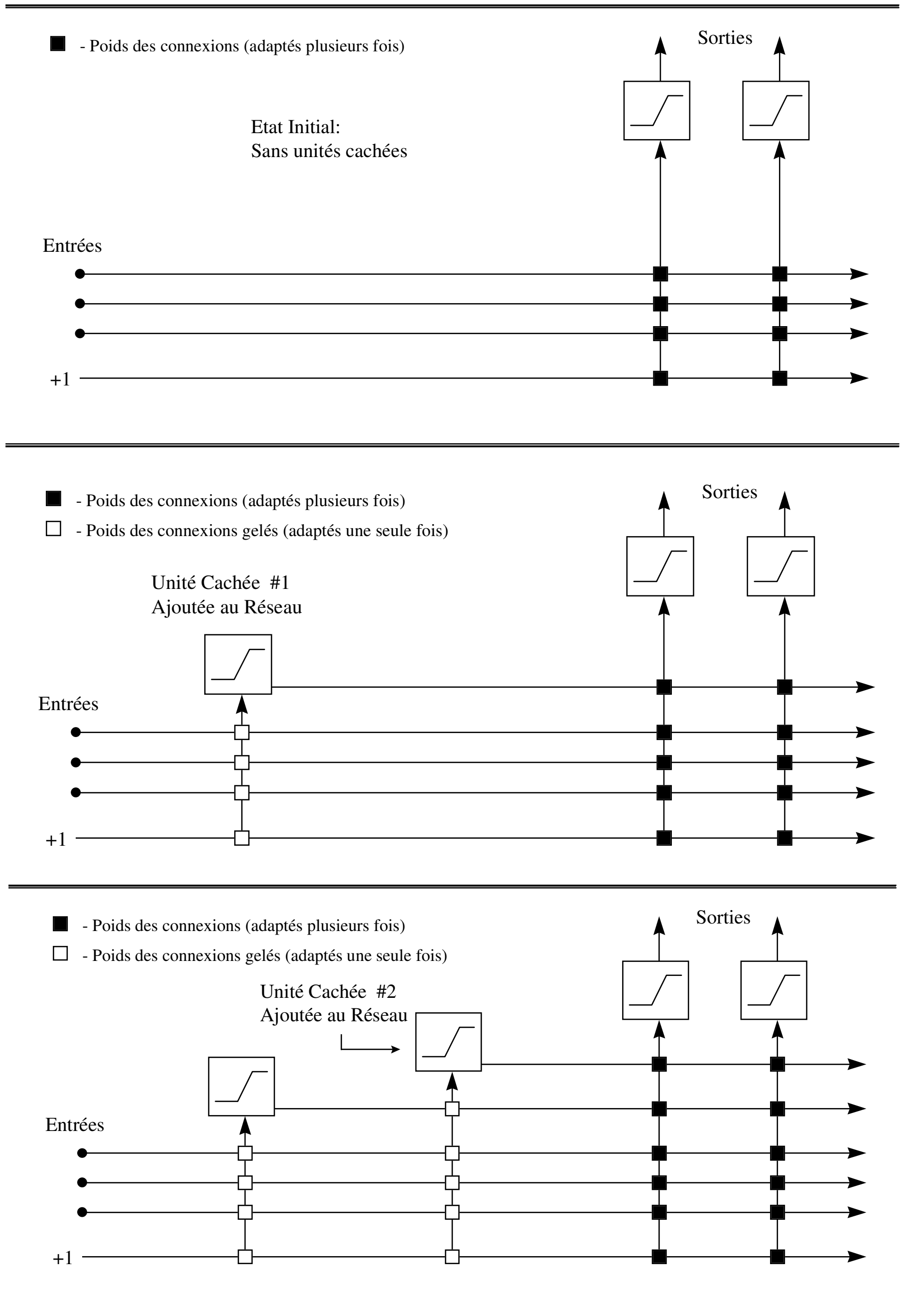

Figure 50 - Architecture des réseaux CasCor (état initial et après l'ajout de deux unités)

Le CasCor commence par l'application de l'algorithme de minimisation de l'erreur des unités de sortie en utilisant la méthode QuickProp [FAH 88]. Cette méthode effectue la minimisation de l'erreur de sortie des unités par rapport aux poids des connexions, en utilisant une méthode de descente de gradient de l'erreur de deuxième ordre [FAH 88] (voir Eq. 27). L'utilisation de cette méthode d'adaptation des poids permet d'augmenter considérablement la vitesse d'apprentissage.

- Quickprop - Adaptation des poids: (Eq. 27)

$$\Delta W(t) = \frac{S(t)}{S(t-1) - S(t)} . \Delta W(t-1)$$

$S(t) = \dfrac{\partial E}{\partial w}$ et $S(t-1) = \dfrac{\partial E}{\partial w}$ dérivés dans le temps courant (t) et précédent (t-1)

Pendant cette étape initiale de l'algorithme CasCor, seuls les poids des unités de sortie sont adaptés. Si l'on atteint un certain niveau d'erreur acceptable, alors l'apprentissage est arrêté et déclaré réussi. Par contre, si l'on ne parvient pas, au bout d'un certain temps prédéterminé (nombre maximal d'époques - *patience*), à minimiser l'erreur résiduelle, alors une nouvelle unité est ajoutée au réseau. L'unité ajoutée reçoit des connexions à partir de toutes les unités d'entrée du réseau et de toutes les unités cachées existantes. Momentanément sa sortie n'est pas reliée au reste du réseau. On adapte alors les poids de ses connexions, en utilisant un algorithme de *maximisation de la corrélation*[1] entre l'erreur résiduelle des unités de sortie et la sortie de l'activation de l'unité ajoutée [FAH 90] (voir Eq. 28). L'algorithme du QuikProp est utilisé cette fois-ci pour adapter les poids de cette unité cachée et ainsi maximiser la covariance entre sa sortie et l'erreur de sortie du réseau. Cette technique de maximisation de la covariance va créer des unités qui seront des détecteurs de traits associés à l'erreur résiduelle du réseau. Une fois que la covariance reste 'stable' pendant un certain nombre d'époques (*patience*), l'unité ajoutée est connectée définitivement au réseau. La sortie de l'unité cachée est connectée aux unités de sorties du réseau avec des poids aléatoires très petits, et les poids d'entrée de cette unité cachée sont "*gelés*". Les poids gelés ne changeront plus et cette unité deviendra un détecteur de traits stable, ce qui évite le problème typique des réseaux à Rétro-Poppagation de l'instabilité des unités (*moving target problem*. - voir section 2.3.3).

---

[1] La fonction employée est en fait la covariance plutôt que la corrélation.

• CasCor - Calcul de la Covariance: (Eq. 28)

$$Cv = \sum_s \left| \sum_e \left(Sa_e - \overline{Sa} \right)\left(Sr_{es} - \overline{Sr_s} \right) \right|$$

Sa_e: Sortie de l' unité ajoutée pour l' exemple ' e'

$\overline{Sa}$: Moyenne de la sortie de l' unité ajoutée sur tous les exemples

Sr_{es}: Sortie ' s' du réseau pour l' exemple ' e'

$\overline{Sr_s}$: Moyenne de la sortie ' s' du réseau sur tous les exemples

Afin d'éviter d'installer définitivement dans le réseau des unités peu utiles, il est possible d'entraîner plusieurs *unités candidates* en même temps. Les unités cachées candidates sont initialisées avec des poids différents, et ainsi on aura plus de chance de trouver un bon ensemble de poids qui va maximiser la covariance. Seule l'unité qui correspond au maximum de covariance sera retenue.

Après la phase d'adaptation de l'unité cachée et de son ajout au réseau, on recommence l'adaptation des unités de sortie, qui désormais incluent les connexions en provenance de la nouvelle unité ajoutée. Ainsi le cycle d'apprentissage pourra être répété plusieurs fois: (1) adaptation des sorties ; (2) ajout et adaptation d'une unité cachée ; (3) gel des poids de la nouvelle unité ; (4) retour à l'étape d'adaptation des unités de sortie. Ce processus est répété jusqu'au succès, c.-à-d. quand l'erreur de sortie est inférieure à une valeur prédéterminée. On peut aussi aboutir à un échec quand on dépasse une limite maximale d'époques sans arriver à réduire suffisamment l'erreur.

L'algorithme CasCor est très performant. Plusieurs études font état de sa performance supérieure à d'autres algorithmes d'apprentissage [FAH 90, THR 91, SCH 93]. L'algorithme CasCor a beaucoup été étudié, et plusieurs variantes de cet algorithme ont été proposées : Cascade-Correlation du Second Ordre (2CCA) [SJO 91], Cascade-Correlation Modifié [SIM 90], Cascade-Correlation Récurrent [FAH 91], et le *'Sibling/Descendant Cascade-Correlation'* (SDCC) [BAL 94]. L'un des seuls reproches qu'on peut faire à cet algorithme est la profondeur importante que peuvent avoir les réseaux obtenus après l'apprentissage, car chaque nouvelle unité est ajoutée dans une nouvelle couche. Dans la pratique, on constate que l'algorithme CasCor converge très rapidement et qu'il n'a pas besoin d'ajouter beaucoup d'unités afin de résoudre un problème. La profondeur finale des réseaux ne représente donc pas un facteur nuisible à son utilisation. Par contre, les réseaux créés avec cet algorithme doivent être utilisés plutôt dans des tâches de classification, car les unités de sortie doivent être discrètes (et les unités cachées le sont aussi). *L'application de ce type d'algorithme dans des tâches d'approximation de fonctions doit être envisagée avec beaucoup de précaution.*

4.3.2 Avantages de l'Utilisation de l'Algorithme Cascade-Correlation

Le CasCor apporte des solutions aux problèmes spécifiques des réseaux PMC basés sur l'algorithme de la Rétro-Propagation, décrits dans la Section 2.3.3, et à d'autres problèmes communs à différents types de réseaux. L'algorithme CasCor permet de :

• Construire le réseau au fur et à mesure que le processus d'apprentissage évolue. Donc, il trouve tout seul la bonne structure du réseau qui est adaptée à une application spécifique.

• Utiliser une méthode d'apprentissage incrémentale qui garde figées les connaissances déjà acquises (poids "gelés"), ce qui permet d'ajouter des nouvelles connaissances à une structure qui a déjà appris un ensemble d'exemples. Cette technique est essentielle puisqu'elle nous permet de faire l'insertion préalable de connaissances et de continuer l'apprentissage à partir d'un certain point, le tout en gardant intactes les connaissances initiales.

• Traiter le problème de la paralysie de l'apprentissage (*flat-spot*), grâce à l'implémentation d'une astuce de l'algorithme QuickProp. Il ajoute une valeur constante à la dérivée de la fonction sigmoïde (*'sigmoid-prime offset'*), ce qui permet d'éviter la paralysie.

• Résoudre en grande partie le problème de l'instabilité et de l'oubli catastrophique (*moving target problem*) des unités du réseau, car une fois qu'une unité devient un 'détecteur spécialisé sur un aspect particulier du problème', elle ne changera plus.

• Réaliser un apprentissage très performant, étant donné l'utilisation d'un algorithme de descente du gradient de deuxième ordre - le QuickProp. Il est beaucoup plus rapide que son prédécesseur, la Rétro-Propagation (*BackProp*).

• Simplifier le choix des paramètres, puisque même si le CasCor nécessite encore un réglage de certains paramètres, l'apprentissage est moins sensible aux changements de leurs valeurs. Les paramètres, tels le pas d'apprentissage (*vitesse*) et l'inertie (*momentum*), ne jouent plus un rôle aussi important (et fort) par rapport au processus d'apprentissage.

De plus, le CasCor possède des propriétés très intéressantes en ce qui concerne la modélisation du développement cognitif chez l'homme. On trouve plusieurs travaux réalisés par T. Shultz [MAR 93, SHU 94a, SHU 94b, BUC 94, SHU 96] qui ont mis en valeur les propriétés du modèle CasCor par rapport à la modélisation du développement cognitif chez l'homme. Ces études nous donnent une motivation de plus pour utiliser la méthode CasCor comme algorithme d'apprentissage constructif de base dans notre système. Dans le chapitre des applications, nous allons revenir sur ce sujet, la modélisation du développement cognitif, concernant le problème de la balance étudié par T. Shultz.

4.3.3 L'Algorithme Cascade-Correlation dans NeuSim

Le réseau obtenu par compilation à partir d'un ensemble de règles sert de point de départ pour l'algorithme CasCor. Les règles sont insérées dans un réseau CasCor d'une manière très simple: tous les poids des connexions sont gelés et le réseau initial est introduit comme étant un ensemble d'unités cachées ajoutées antérieurement par l'algorithme CasCor. Le simulateur NeuSim nous permet le choix entre deux possibilités différentes d'insertion d'un réseau pré-construit :

1. Insertion de tout le réseau, où toutes les unités restent figées. Dans ce cas de figure, nous avons besoin de créer une nouvelle couche de sortie afin de préserver inaltérées les unités de sortie du réseau initial. La Figure 51 montre un exemple de cette situation, qui est d'ailleurs le mode de fonctionnement de NeuSim que nous avons le plus étudié.

2. Insertion de tout le réseau, où presque toutes les unités restent figées, à l'exception des unités de sortie. Dans ce cas de figure, les unités de sortie du réseau initial sont utilisées comme des unités de sortie du réseau CasCor, et par conséquent elles sont postérieurement modifiées pendant le processus d'apprentissage. Cette approche nous oblige à considérer que les unités de sortie (conclusions finales de la base de règles) pourront être modifiées. Un réseau ainsi créé est moins complexe qu'un autre créé en utilisant la méthode décrite au paragraphe précédent, mais on ne peut plus considérer que les nouvelles connaissances acquises par apprentissage restent totalement séparées des connaissances de départ.

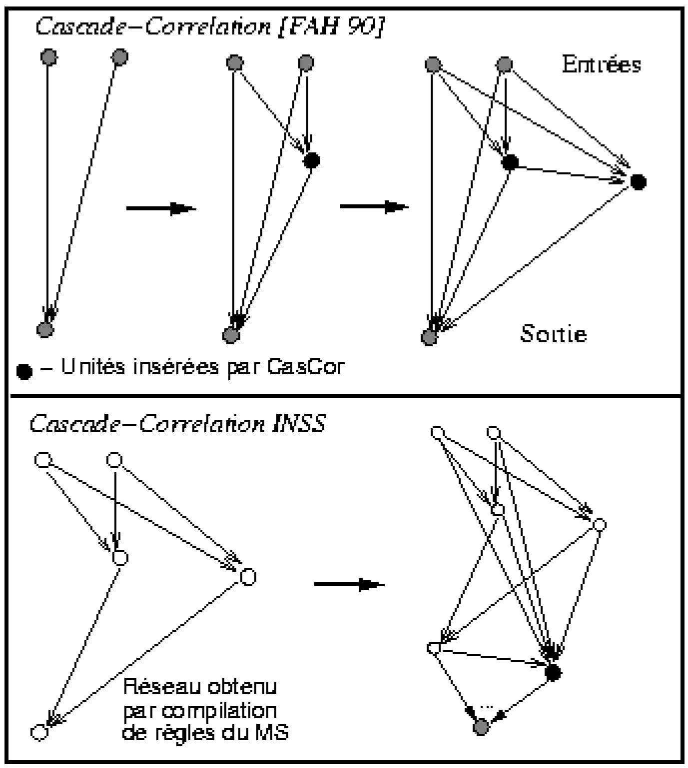

Figure 51 - L'algorithme Cascor et l'évolution de la structure d'un réseau dans INSS

Le simulateur NeuSim offre plusieurs options de configuration, parmi lesquelles on en trouve une qui permet de sélectionner le mode d'insertion d'un réseau initial que nous venons de décrire. D'autres options configurables liées à l'algorithme CasCor sont : la valeur de la patience pour les unités de sortie, la valeur de la patience pour les unités cachées et le nombre d'unités candidates à utiliser pendant l'apprentissage.

4.3.4 Le Module NeuSim en tant que simulateur de réseaux

Il s'avère que le module NeuSim, développé dans le cadre du système INSS et de la méthode connexionniste CasCor, peut être utilisé de manière plus générale comme un simulateur d'autres types de RNAs à base de PMC. Il offre la possibilité de simuler des réseaux avec l'une de ces trois méthodes d'apprentissage : Rétro-Propagation, QuickProp ou CasCor.

NeuSim est un simulateur très flexible qui accepte différents réglages des paramètres de simulation, et qui possède les propriétés nécessaires pour pouvoir être intégré dans le cadre du système hybride neuro-symbolique INSS. Il possède une interface très simple et non graphique, ce qui rend facile son utilisation et aussi son port sur d'autres machines. La simulation peut être faite de façon interactive (interaction par menus) ou de façon plus automatique (utilisation de fichiers de configuration).

Nous allons donner ici un bref aperçu des paramètres de configuration du simulateur et de son fonctionnement. Quand il est en mode d'apprentissage, NeuSim utilise trois fichiers principaux d'entrée : un fichier *.learn (données d'apprentissage supervisé incluant la valeur de la sortie désirée), un fichier *.test (données de test de généralisation), et un fichier *.cfg (spécifie les paramètres et le déroulement de la simulation). Quand il est en mode de rappel, le fichier *.learn n'est pas utilisé. Deux autres fichiers importants sont les fichiers *.top et *.wts, qui décrivent la topologie et les poids respectivement d'un réseau préalablement construit. Ces deux fichiers peuvent être utilisés en mode d'apprentissage (connaissances de départ), seulement si la méthode d'apprentissage choisi est le CasCor. Les fichier *.top et *.wts sont indispensables quand on passe en mode de rappel (sauf si le réseau vient d'être créé et entraîné). Nous présentons des exemples de tous ces fichiers dans l'Annexe B. La Figure 52 présente un schéma des fichiers d'entrée et de sortie utilisés par NeuSim.

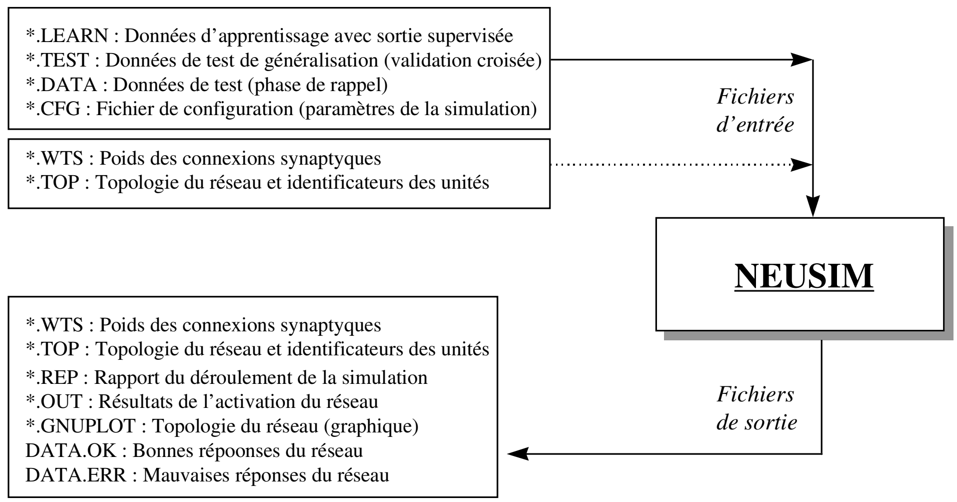

Figure 52 - Fichiers utilisés par NeuSim

Le fichier de configuration de NeuSim contient les informations suivantes (nous listons à côté les principaux paramètres décrits dans ces fichiers - reportez-vous à l'Annexe B pour obtenir plus d'informations sur ces paramètres) :

- *La définition du type de réseau utilisé, de sa structure, et des ses unités* : le nombre d'entrées (NInputs), le nombre de sorties (NOutputs), le choix des valeurs aléatoires (RndRange, RndSeed), le type de fonction d'activation (FunctActv, SigmTemp), le type de sortie (OutType, OutNorm);

- *La définition du type d'algorithme d'apprentissage et de ses paramètres respectifs* : la méthode d'apprentissage (Learning - BackProp, QuickProp ou CasCor), le nombre d'unités cachées (BackProp - réseau à trois couches), le nombre d'unités candidates (CasCor), le pas d'apprentissage (Epsilon), l'inertie (Momentum), les critères d'arrêt (StopCrit, StopErr, MaxEpochs), la façon et l'ordre de présentation des données (EpLearn, DataOrder), le type de sortie utilisée pour évaluer le nombre d'erreurs (NetOutput, MaxErr), 'sigmoid prime offset' (SPOffset), 'weight decay' (WtsDecay), la patience du CasCor (CCOutPat, CCOutECh, CCHidPat, CCHidECh), l'ajout d'une couche de sortie supplémentaire (CCAddOut);

- *La définition du type de suivi de la simulation à enregistrer dans le fichier de rapport et à présenter à l'utilisateur* : le type d'information et la fréquence d'enregistrement dans le fichier de rapport (RepLevel, RepFreq, ErrLevel - fichier *.rep), le type d'information et la fréquence d'affichage d'information à l'utilisateur (UserRep, UserFreq), l'origine des données utilisées pour créer le fichier de sortie avec les réponses du réseau (OutFile - fichier *.out);

- *La définition des principales tâches à réaliser pendant la simulation* (Task);

```
%%
%% Fichier de Configuration de NeuSim
%% Fichier Xor.Cfg - Apprentissage de la fonction XOR avec
%%                   la méthode Cascade-Correlation
%%
Task      LT       % Tâche: Réaliser l'apprentissage ('L') et après
                   %        le test final de généralisation ('T')
MaxEpochs 1500     % Nombre maximale d'époques
NInputs   2        % Nombre d'entrées du réseau
NOutputs  1        % Nombre de sorties du réseau
Learning  1        % Méthode d'apprentissage: Cascade-Correlation
Cascor    8        % Utiliser CasCor avec 8 unités candidates
CCOutPat  16       % Patience - Unités de sortie (# époques)
CCOutECh  0.01     % Patience - Unités de sortie (erreur)
CCHidPat  16       % Patience - Unités cachées (# époques)
CCHidECh  0.03     % Patience - Unités cachées (erreur)
EpLearn   1        % Apprentissage par paquets (par époques)
RndRange  0.001    % Valeurs aléatoires dans l'intervalle +/- 0.001
Epsilon   0.0001   % Pas d'apprentissage (vitesse de convergence)
MaxErr    0.4      % Sortie binaire - erreur acceptée = 0.4
StopCrit  1        % Arrêt avec 100% de réponses correctes
SPOffset  0.1      % Valeur du 'Sigmoid prime offset'
RepLevel  128      % Présenter les résultats finaux
UserRep   1        % Activer la sortie des résultats sur l'écran
%% Fin
```

Figure 53 - Exemple d'un fichier de configuration de NeuSim

Le paramètre du fichier de configuration qui spécifie les tâches à réaliser est un élément très important de notre simulateur. C'est à travers de ce paramètre qu'on va identifier la séquence d'opérations à réaliser. Les tâches réalisées par le simulateur peuvent être les suivantes :

• Charger un réseau ('P') : le simulateur va réaliser la lecture des informations qui décrivent un réseau pré-construit, pour ensuite l'utiliser afin de le raffiner par apprentissage, ou de simplement réaliser une consultation (mode de rappel). Cette option va charger les deux fichiers qui décrivent le réseau, le fichier <net>.wts et le fichier <net>.top;

• Créer un réseau ('R') : le simulateur va créer un réseau d'après la description de la topologie indiqué dans le fichier <net>.top. Les poids des connexions vont ensuite être initialisés avec des valeurs aléatoires;

• Réaliser l'apprentissage ('L') : le simulateur va lancer l'algorithme d'apprentissage du réseau (selon la méthode choisie) en utilisant le fichier de données <data>.learn. L'algorithme d'apprentissage va s'arrêter dans un des cas suivants : après avoir dépassé un maximum d'époques d'apprentissage, après avoir obtenu 100% de bonnes réponses sur le fichier d'apprentissage, après avoir obtenu 100% de bonnes réponses sur le fichier de test de généralisation, ou après avoir réduit l'erreur de sortie jusqu'à une valeur inférieure à la valeur définie dans le fichier de configuration. La sélection de la méthode d'arrêt à employer est aussi indiquée dans le fichier de configuration;

• Réaliser des tests de généralisation ('G') : le simulateur fait un test de généralisation (fichier <data>.test) après chaque époque d'apprentissage (fichier <data>.learn). Chaque époque est donc constituée par : une présentation de toute la base d'apprentissage, l'adaptation des poids des connexions, et une présentation des données de test de généralisation;

• Réaliser un test final ('T') : le simulateur réalise un test sur les données d'un fichier spécifié comme le fichier de test. Cela se produit soit après la phase d'apprentissage (si l'option 'L' est indiqué), ou directement après le chargement d'un réseau pré-construit;

• Enregistrer les sorties du réseau ('O') : le simulateur va créer un fichier <data>.out contenant les réponses données par le réseau lors de son activation. Cette option doit être utilisée avec l'option 'T';

• Enregistrer séparément les bonnes et mauvaises réponses ('E') : le simulateur va créer deux fichiers de sortie, un fichier <data>.ok contenant les données qui ont été classées correctement, et un fichier <data>.err contenant les données qui n'ont pas été classées correctement. Les données utilisées dans ce test sont les données de test de généralisation. Cette option nous offre la possibilité d'analyser les 'cas problématiques' et de mieux valider le problème en question;

• Compter les réponses fausses ('F') : le simulateur présente les totaux des bonnes et des mauvaises réponses du réseau. Ces résultats sont basés uniquement sur des valeurs de sortie du type binaire (les valeurs continues sont discrétisées). Il va classer les réponses dans quatre groupes : sortie désirée vraie avec une sortie du réseau fausse, sortie désirée fausse avec une sortie du réseau vraie, sortie du réseau correcte, et sortie du réseau dans l'intervalle défini comme 'sans réponse'. Cette option nous permet de mieux interpréter les erreurs de réponse du réseau qui sont divisées en trois groupes (faux-0, faux-1, indéterminé);

• Enregistrer les poids du réseau ('W') : une fois terminé l'exécution des autres tâches, le simulateur va enregistrer les poids des connexions du réseau dans un fichier (<net>.wts). Cette option nous permet de reprendre l'apprentissage ou l'activation du réseau plus tard;

• Enregistrer la topologie du réseau ('S') : une fois terminée l'exécution des autres tâches, le simulateur va enregistrer la topologie du réseau dans un fichier (<net>.top);

• Enregistrer dans un fichier graphique la topologie du réseau ('V') : une fois terminé l'apprentissage, le simulateur va créer un fichier compatible avec Gnuplot contenant une description graphique de la topologie finale du réseau. Cette option nous permet de visualiser la topologie du réseau utilisée par le simulateur;

• Activer le menu ('I') : le simulateur, après avoir exécuté toute autre tâche définie dans le fichier de configuration, passe dans un mode d'interaction par menus. L'utilisateur peut ainsi exécuter toute une série d'opérations réalisées de façon automatique et ensuite passer dans un mode interactif afin de réaliser des tâches complémentaires.

Une des options présentes dans la liste de tâches, l'option 'G', nous permet de trouver le point optimal dans la courbe d'évolution de l'erreur de généralisation (voir Figure 10). Cette option est très intéressante puisqu'elle nous permet de trouver la meilleure configuration du réseau obtenue pendant tout le processus d'apprentissage. La simulation est faite de la façon suivante :

1. Après chaque époque d'apprentissage, le réseau est activé avec la base d'exemples de test afin de mesurer l'erreur de généralisation;

2. Le simulateur doit poursuivre l'apprentissage même en dépassant le point optimal, car on n'a pas les moyens de l'estimer par avance;

3. Chaque mesure de l'erreur de généralisation est comparée avec l'erreur de l'époque précédente. Si l'erreur actuelle est inférieure aux erreurs précédentes, le simulateur va garder cette valeur et l'indication de l'époque dans laquelle elle s'est produite;

4. A la fin de la simulation on aura l'indication de l'époque ('BestEpoch') où s'est produite la plus petite erreur de généralisation ('BestScore') de toute l'évolution du processus d'apprentissage;

5. Etant donné que le simulateur permet de configurer la valeur qui contrôle la génération des nombres aléatoires ('RndSeed'), il suffit de la régler exactement avec la même valeur utilisée dans une simulation précédente pour qu'on puisse avoir exactement la même évolution du réseau

pendant le processus d'apprentissage. La valeur de 'RndSeed' utilisée dans la dernière simulation peut être retrouvée dans le fichier de rapport (*.rep);

6. Par conséquence, si l'on veut arrêter le processus d'apprentissage à l'époque correspondant à la meilleure généralisation, il suffit d'utiliser la même valeur de 'RndSeed' utilisée lors d'une simulation précédente puis d'arrêter l'apprentissage à l'époque qui a permis le meilleur score lors de cette simulation précédente. (On donne à 'MaxEpochs' la valeur 'BestEpoch' antérieure).

Ce type de procédure nous permet de mieux exploiter les capacités d'apprentissage des réseaux afin d'obtenir le réseau le plus intéressant possible : celui qui donne la meilleure généralisation par rapport aux données disponibles.

Pour conclure la discussion sur les options de configuration, il est intéressant de souligner que le simulateur permet d'évaluer le pourcentage de bonnes réponses selon un critère prédéterminé par l'utilisateur. L'utilisateur doit indiquer l'erreur maximale ('MaxErr') qu'il acceptera par rapport aux valeurs de sortie du réseau, car même si ces unités donnent des valeurs continues comme résultat, la plupart du temps on cherche à interpréter les résultats obtenus dans les sorties comme des valeurs discrètes (le réseau est utilisé dans des tâches de classification). Ainsi, une sortie avec une valeur égale à X est reconnue comme 'correcte', si X est compris dans l'intervalle donné par la valeur de sortie désirée plus/moins une erreur (intervalle [Sd-MaxErr, Sd+MaxErr]). Par exemple, si l'on fixe MaxErr=0.3 et étant donné la sortie désirée égale à 1.0, on va considérer les réponses du réseau comme incorrectes pour toute valeur de sortie inférieure à 0.7. Donc, la valeur spécifiée pour le paramètre MaxErr joue un rôle très important par rapport à l'évaluation des erreurs commises par le réseau, ainsi que pour la définition de la précision des réponses exigée par l'utilisateur.

4.3.5 Discussion sur le Module Connexionniste

Nous avons présenté ici une description des principales caractéristiques et du mode de fonctionnement du module connexionniste ***NeuSim*** qui constitue une partie très importante de notre système hybride. Ce module présente des caractéristiques très intéressantes, telles que : il est flexible ; il est performant (méthode de descente de la courbe d'erreur de deuxième ordre) ; il est incrémental (adaptation des poids et aussi de la topologie du réseau) ; il permet d'utiliser des connaissances théoriques compilées sous la forme d'un réseau ; il ne mélange pas les connaissances de départ avec les nouvelles connaissances ; il ne modifie pas la signification des unités après l'apprentissage ; il permet d'ajouter des nouvelles unités afin de corriger les problèmes existants dans les connaissances insérées dans le réseau. De plus, notre simulateur offre des possibilités très intéressantes de configuration et d'accompagnement des simulations, ainsi que des fonctions d'aide à la validation des résultats fournis par le simulateur.

Le grand nombre de paramètres réglables disponibles ne pose pas de difficultés supplémentaires aux utilisateurs, au contraire de ce qu'on pourrait imaginer. Pour simplifier encore plus la tâche de création d'un fichier de configuration, le simulateur NeuSim possède des fichiers modèles, lesquels peuvent être appliqués à d'autres problèmes sans avoir besoin de régler tous les paramètres. Une nouvelle application ne nécessite pas un expert pour réaliser la configuration du réseau, tout pouvant être facilement réglé à partir des fichiers modèles. D'un autre côté, tous ces paramètres vont permettre l'utilisation de ce simulateur dans un grand nombre de situations, où l'utilisateur peut les adapter selon ses besoins spécifiques.

NeuSim s'est avéré un outil facilement adaptable à des applications très spécifiques. Tel est le cas d'une interface Web (accès via l'Internet) de consultation à un système d'aide au diagnostic médical que nous avons développée (voir Section 5.2) ; et aussi d'un système réactif de contrôle d'un petit robot autonome - Khepera (voir Section 5.4). Finalement, ce simulateur sert aussi à réaliser des tâches de validation de connaissances et il s'intègre parfaitement à notre système INSS, d'une façon telle que nous allons pouvoir, dans la suite, 'ouvrir la boîte noire' que constitue le réseau et en extraire des règles propositionnelles.

4.4 EXTRACTION DE CONNAISSANCES

Le module Extract permet d'obtenir des règles symboliques à partir d'un réseau créé par INSS. Nos méthodes d'extraction de règles à partir des réseaux sont basées sur les algorithmes employés avec les réseaux KBANN (SUBSET et NofM - voir Section 3.3.2.3) auxquels nous avons apporté des améliorations. Ces algorithmes sont des méthodes du type décompositionnel. Ils ont été implémentés par Loïc Decloedt dans le cadre de son stage de D.E.A. [DEC 95] réalisé sous la supervision de F. Osório. Nous avons cherché a développer un outil d'extraction de règles, en nous donnant comme buts principaux :

- Obtenir une réduction de la complexité du processus d'extraction. L'extraction de règles par des méthodes comme SUBSET ou NofM peut demander un très grand effort computationnel selon le nombre d'unités et de connexions de chaque unité. Des expériences pratiques montrent que l'extraction de toutes les connaissances d'un réseau complexe peut être impraticable, et qu'il faut donc essayer de simplifier ce processus. Nous avons choisi de réaliser une simplification du réseau avant de réaliser l'extraction;

- Augmenter l'intelligibilité des règles. Un très grand nombre de règles peut être difficile à interpréter. Il faut trouver des moyens de représenter les connaissances d'une façon simple et synthétique;

- Réaliser une extraction sélective de règles. Les règles représentent une généralisation des connaissances décrites dans la base de cas. Il faut donc trouver tout d'abord les règles les plus générales possibles;

- Offrir la possibilité de valider les connaissances acquises et déjà représentées sous la forme de règles symboliques. Il faut pouvoir extraire des connaissances, mais pour qu'elles soient utiles, il faut les valider. Rappelons que nous partons du principe que ni les connaissances théoriques, ni les connaissances empiriques ne sont complètement correctes.

Le module Extract que nous avons conçu est un outil capable d'obtenir des règles symboliques (fichier *.symb) à partir d'un réseau préalablement entraîné (fichiers *.wts et *.top). Pour ce faire, nous réalisons tout d'abord une simplification du réseau composée d'une étape de sélection d'unités et d'une étape de sélection de connexions. Une fois simplifié le réseau, nous allons appliquer l'algorithme d'extraction SUBSET afin d'obtenir les règles symboliques. Nous avons implémenté aussi la méthode NofM, mais cette méthode n'a pas été utilisée dans le cadre des expériences pratiques développées dans cette thèse.

La méthode d'extraction de règles que nous avons mise au point et qui utilise une simplification préalable du réseau, a été nommée ***RULE_OUT*** [DEC 96].

4.4.1 Sélection des Unités pour l'Extraction de Règles

Nous avons mis à la disposition de l'utilisateur trois méthodes de sélection des unités pour l'extraction de règles. Ces méthodes permettent de réduire significativement la quantité globale d'informations (unités) à analyser. Nous arrivons ainsi à simplifier notablement le processus d'extraction de règles. Les trois méthodes sont les suivantes :

1. *Sélection des unités qui représentent les nouvelles connaissances acquises.* Compte tenu de ce que INSS offre la possibilité d'intégrer des connaissances théoriques dans un réseau par compilation de règles, et que le processus d'apprentissage permet de les préserver inaltérées, nous n'avons pas besoin d'extraire à nouveau les connaissances théoriques initiales. Nous allons réaliser l'extraction des connaissances seulement sur les unités qui ont été modifiées (les unités ajoutés par l'algorithme CasCor) et qui représentent les nouvelles connaissances acquises. Ainsi, nous allons obtenir un ensemble de règles beaucoup plus compact contenant seulement les règles qui représentent les changements du réseau par rapport aux connaissances de départ. La sélection des unités modifiées par le processus d'apprentissage est très simple puisque nous les étiquetons lors de leur création. Il suffit alors de ne pas extraire des règles à partir des unités 'gelées' du réseau, ce qui permet de réduire significativement le temps de calcul. De plus, cette sélection d'unités nous permet d'implémenter une méthodologie d'extraction incrémental de connaissances. Ainsi, notre méthode d'extraction de règles s'intègre parfaitement dans le concept que nous proposons d'un système hybride d'apprentissage constructif.

2. *Sélection des unités d'entrées les plus significatives.* Nous proposons à l'utilisateur la possibilité de réaliser un prétraitement du réseau afin de réduire encore plus la quantité d'informations à traiter lors de l'extraction de règles. Nous avons implémenté la méthode d'élagage d'unités d'entrée du réseau basée sur l'analyse de sensibilité proposée par J. Moody [MOO 94] (*Sensitivity-based prunning method for input units - SBP algorithm*). Cette méthode est basée sur le calcul d'une 'mesure de sensibilité' S_i, qui va évaluer le changement infligé sur l'erreur d'apprentissage étant donné l'élagage d'une entrée X_i du réseau. Plus précisément, la mesure S_i (voir Eq. 29) indique l'effet du remplacement de l'entrée X_i par sa valeur moyenne (calculée sur tous les exemples de la base d'apprentissage) sur l'erreur quadratique (SE).

$$Si = \frac{1}{N}\sum_{j=1}^{N}\frac{\partial SE(j)}{\partial x_i(j)} \quad .dx_i(j) \qquad \text{(Eq. 29)}$$

$$d\,x_i(j) = \left(x_i(j) - \bar{x}_i\right)$$

N : nombre d'exemples dans la base d'apprentissage

$\bar{x}_i$: valeur moyenne de l'entrée X_i

$x_i(j)$: valeur de l'entrée 'i' pour l'exemple 'j' de la base d'apprentissage

Le module Extract fait le calcul de l'influence des entrées sur l'erreur globale, offre la possibilité de les afficher à l'écran, et propose à l'utilisateur d'établir un seuil d'influence. Le système coupe alors toutes les connexions vers ces unités d'entrée (élimination des entrées) dont l'influence est inférieure au seuil considéré.

3. *Sélection des unités du réseau qui ont le plus d'influence sur le comportement du réseau.* Nous avons développé une méthode empirique de simplification du réseau, qui consiste à détecter les unités qui ont une valeur de sortie d'activation pratiquement constante pour tous les exemples contenus dans une base de connaissances indiquée par l'utilisateur. Ces unités peuvent donc être approximées par des unités à sortie constante, c'est-à-dire, avec une sortie toujours égale à 0.0 ou 1.0. Ensuite, nous pouvons les éliminer en changeant le seuil des unités auxquelles elles sont connectées [DEC 95]. De cette façon, nous avons la possibilité de simplifier le réseau sans affecter significativement sont comportement.

C'est l'utilisateur qui va choisir le seuil d'influence à partir duquel les unités vont être éliminées. Le seuil va être indiqué par la valeur de la fréquence d'activation des unités à considérer (voir Figure 54).

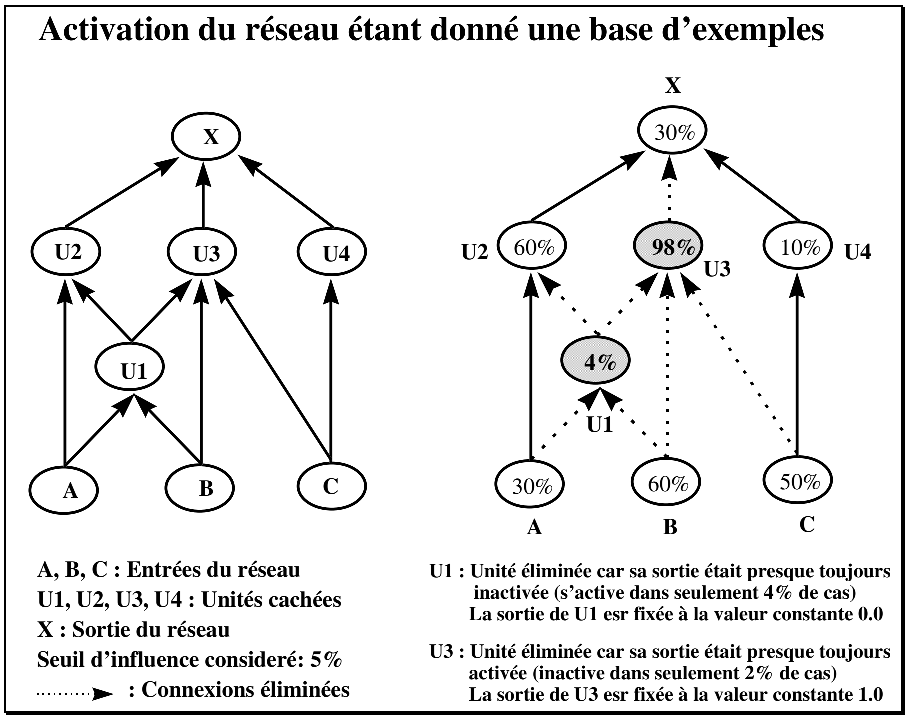

Figure 54 - Sélection des unités basée sur la pourcentage d'activation

La fréquence d'activation d'une unité indique le pourcentage des cas de la base de d'exemples pour lesquels la sortie de cette unité est égale à 1.0 (activée). Si la base d'exemples couvre bien l'espace d'entrées et si les exemples sont bien repartis dans les classes (dans le cas de la Figure 54, les classes sont X=0.0 et X=1.0), alors la fréquence d'activation peut être un bon paramètre pour estimer l'influence d'une unité sur les réponses données par le réseau. Le module Extract laisse à l'utilisateur le choix de l'application de cette méthode de simplification du réseau ainsi que de la valeur du seuil d'influence à utiliser.

Le module Extract offre une interface à l'utilisateur basée sur des menus, qui lui permet de définir jusqu'à quel point il veut simplifier le réseau. L'interface lui offre le choix d'application des différentes méthodes que nous présentons ici, et en plus, il a la possibilité de mesurer à tout moment l'influence d'une simplification sur l'erreur des réponses du réseau par rapport à une base d'exemples donnée. Si l'utilisateur ne veut pas réaliser la simplification du réseau avant l'extraction, il peut passer directement à l'application de la méthode d'extraction. Dans ce dernier cas, le processus d'extraction va être beaucoup plus lent, demander plus de ressources (mémoire) de la machine, et doit conduire à l'obtention d'un ensemble de règles beaucoup plus grand.

4.4.2 Sélection des Connexions pour l'Extraction de Règles

Nous avons aussi développé des méthodes pour mesurer l'influence des poids par rapport aux sorties du réseau, de façon à pouvoir éliminer celles qui ont le moins d'influence. La possibilité de réaliser cette simplification du réseau est très importante, car la complexité du processus d'extraction est fortement liée au nombre de connexions d'entrée de l'unité analysée. Le nombre de règles obtenues par extraction peut lui aussi être réduit de façon significative. Nous avons mis à la disposition de l'utilisateur deux méthodes de sélection des connexions pour élaguer le réseau avant l'extraction de règles. Les deux méthodes sont les suivantes :

1. *Sélection des connexions les plus faibles*. Nous avons développé une méthode empirique de simplification des connexions, basée sur ce que nous avons appelé l'étude du pourcentage des poids. Cette méthode est basée sur la supposition que, parmi les connexions qui arrivent à une même unité, celles qui ont les poids les plus faibles (valeurs très petites) par rapport aux autres, sont les connexions qui ont le moins d'influence sur la sortie de l'unité considérée. Cette hypothèse s'est avérée valable dans nos expériences pratiques portant sur les réseaux créés avec l'algorithme CasCor. L'algorithme CasCor semble bien attribuer des poids très faibles aux connexions moins importantes.

La méthode que nous avons développée consiste à mesurer l'importance relative de chaque poids par rapport aux autres poids de même signe liés à la même unité [DEC 96]. Pour faire cela, nous calculons le pourcentage de chaque poids par rapport à la somme globale des autres poids (positifs et négatifs). Ainsi, l'utilisateur peut spécifier un seuil d'influence (pourcentage) utilisé pour la sélection des poids à éliminer. Il pourra aussi consulter l'effet de cet élagage sur l'erreur globale d'activation du réseau.

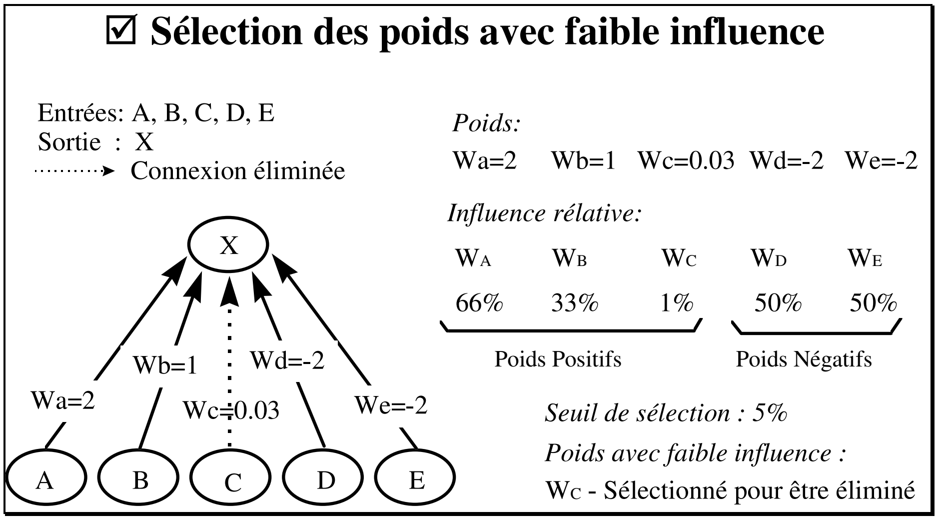

Figure 55 - Sélection des connexions les plus faibles

2. *Sélection des connexions par rapport à leur influence sur l'erreur globale*. La supposition faite que l'influence des connexions est proportionnelle à la valeur relative des poids ne prend pas en compte un autre facteur important : si un poids très petit a peu d'influence sur la sortie de l'unité, par contre il a été aussi observé que parfois des connexions avec des poids assez grands n'avaient pas un rôle significatif par rapport aux réponses globales du réseau. Ainsi, nous avons implémenté une autre méthode inspirée par l'algorithme OBD (*Optimal Brain Damage*) [LEC 90], qui lui va prendre en compte cet aspect. Cet algorithme a été développé afin de mesurer l'influence de chaque poids sur l'erreur global, à travers le calcul de la dérivée seconde de l'erreur globale. Cette mesure de l'influence d'un poids Wk sur l'erreur quadratique moyenne ASE (*Averege Squared Error*) d'activation d'un réseau, appelée *'Saliency'*, est définie par l'équation suivante :

$$S(w_k) = \frac{1}{2}\frac{\partial^2 ASE}{\partial w_k \partial w_k}.(w_k)^2 \qquad \text{(Eq. 30)}$$

Les poids avec une faible 'Saliency' peuvent donc être éliminés du réseau, et cela va permettre à l'utilisateur d'optimiser encore plus le processus d'extraction de règles.

4.4.3 Application de l'algorithme SUBSET dans INSS

Une fois terminée la sélection des unités et des connexions qui vont être considérées pour l'extraction de règles, nous passons à l'application de l'algorithme SUBSET. L'algorithme SUBSET réalise l'extraction de règles d'ordre 0 à partir du réseau simplifié, selon les procédures que nous avons décrites dans la section 3.3.2.3. Il n'y a pas de différence significative entre l'implémentation de SUBSET dans INSS et la version d'origine utilisée dans KBANN. La différence entre l'approche KBANN et notre approche se trouve dans le post-traitement qu'on peut appliquer aux règles extraites. Ainsi, le module Extract met à disposition de l'utilisateur des procédures de simplification du réseau avant extraction, d'extraction de règles par l'algorithme SUBSET, et de simplification de la base de règles extraite à partir du réseau.

Notre méthode de post-traitement des règles extraites consiste à calculer la représentativité de chaque règle par rapport à une base d'exemples fournie par l'utilisateur. Nous obtenons une mesure qui indique le pourcentage des exemples qui activent une règle spécifique (le nombre de fois que ses antécédents sont satisfaits). Cette mesure est alors présentée à l'utilisateur afin qu'il puisse se décider sur le niveau de simplification à appliquer sur cette base de règles obtenue par l'algorithme SUBSET.

La justification d'une telle approche est qu'il peut arriver que certaines règles ne soient pas utiles dans un contexte particulier, car elles ne sont pas applicables dans la pratique. Par exemple, prenons le cas d'un réseau possédant deux entrées booléennes E1 et E2 dont les valeurs sont *toujours* complémentaires. Il peut arriver que le système d'extraction obtienne en résultat une règle dont la condition soit une conjonction de ces entrées : *Si E1 et E2 alors X*. Cette règle peut être valide, mais elle ne peut s'appliquer dans le cas présent, car sa condition ne sera jamais valide. L'inconvénient est que, si l'on obtient un grand nombre de règles de ce type, on risque de se retrouver avec beaucoup de règles inutiles qui risquent alors de diminuer la compréhensibilité du résultat. Pour pallier à ce problème, l'utilisateur peut faire appel à la méthode de post-traitement que nous venons de décrire. En conséquence, nous sommes capables alors d'éliminer les règles inutiles qui ne sont jamais, ou presque, utilisées dans la pratique, et d'augmenter ainsi la compréhensibilité du résultat.

Cette fonctionnalité est plus importante qu'il y paraît au premier abord, car le nombre de règles inutiles de ce type peut être très grand. En effet, si l'on modélise un problème dont un des paramètres est une variable discrète (type 'nominal') pouvant prendre un nombre de valeurs finies, alors on va créer une unité d'entrée pour chaque valeur possible de ce paramètre (codage adopté par INSS pour les variables discrètes). En pratique, les valeurs de ces unités s'excluent mutuellement, c'est-à-dire qu'une et une seule de ces unités peut être activée à cet instant donné. Or, ce type d'informations n'apparaît pas au niveau du réseau, et on traite toutes les entrées de manière équivalente. Donc, le système d'extraction, lors de la recherche des règles possibles, va étudier les combinaisons de ces entrées qui s'excluent mutuellement, et peut donc en déduire des règles tout à fait correctes dans l'absolu, mais qui ne seront jamais utilisées. Par exemple :

Variable - Couleur : nominal # 3 : rouge, verte, bleue;

Règles extraites - informations redoublés :

x := couleur(rouge), not(couleur(verte)), not(couleur(bleue));

x := couleur(rouge), not(couleur(verte));

x := couleur(rouge), not(couleur(bleue));

x := not(couleur(verte)), not(couleur(bleue));

Règles extraites - informations invalides :

x := couleur(rouge), couleur(verte), couleur(bleue);

x := couleur(rouge), couleur(verte);

x := couleur(rouge), couleur(bleue);

x := couleur(verte), couleur(bleue);

Ce nombre de règles parasites peut devenir très élevé si un paramètre de départ discret peut prendre beaucoup de valeurs. Dans certains cas, le module d'extraction finit par trouver des méta-règles, qui n'expriment pas des connaissances sur le problème en soi, mais des connaissances sur les connaissances (liées à la forme de codage et à la validité des variables).

4.4.4 Discussion sur le Module d'Extraction de Règles

Nous avons présenté ici la méthode d'extraction de règles ***RULE_OUT***, capable de réaliser une simplification préalable des réseaux afin de réduire la complexité du processus d'extraction de règles. La méthode *RULE_OUT* est une amélioration de la méthode SUBSET d'extraction de règles, cependant cette méthode peut aussi être utilisée avec d'autres algorithmes décompositionnels d'extraction de règles, tel est le cas de l'algorithme NofM. Notre méthode s'intègre parfaitement au système INSS, car elle est une méthode d'extraction incrémentale de connaissances. L'utilisation de *RULE_OUT* nous permet d'obtenir juste les nouvelles connaissances acquises, tout en préservant inaltérées les connaissances de départ.

Les méthodes d'extraction de règles avec l'élagage des réseaux que nous avons décrites dans les sections précédentes, ont été testées sur différents problèmes (e.g. les problèmes du Moine - "Monk's problem", voir Section 5.1) et comparées avec l'extraction de règles sans simplification des unités et des connexions. Les résultats obtenus montrent bien une réduction très sensible de la complexité du processus d'extraction après la simplification du réseau, ainsi qu'une amélioration de la qualité des règles obtenues (augmentation de la compréhensibilité par la réduction du nombre de règles obtenues) [DEC 95]. Nous allons présenter les résultats de l'utilisation du module d'extraction de règles dans le chapitre qui traite des applications du système INSS (Chapitre 5), dont les problèmes du Moine [THR 91] composent une des applications qui nous a permis de bien mettre en valeur les principales propriétés de notre implémentation.

Les travaux que nous avons développés sur ce sujet, nous ont amené à poser un certain nombre de questions à propos de la représentation de connaissances. En réalité, un des nos principaux buts a depuis toujours été le développement d'outils pour la manipulation et la conversion des différents types de représentation de connaissances : la transformation de règles en réseaux, l'acquisition et la représentation de connaissances empiriques par des réseaux, l'extraction de connaissances des réseaux (on devrait plutôt dire 'explicitation de connaissances'). Ces différents processus servent à manipuler les connaissances afin de les transformer d'une forme de représentation à l'autre (e.g.: symbolique - règles propositionnelles, symbolique - base d'exemples, numérique - connexions et poids). Les questions qui se posent sont donc : pourquoi doit-on choisir une méthode de représentation plutôt qu'une autre ? Existe-t-il une méthode de représentation plus puissante que toutes les autres ? Quelles sont les limitations de la capacité de représentation d'une méthode spécifique ? Nous n'avons pas l'intention de répondre précisément à ce type de questions dans cette thèse, car nous pensons que çe serait trop ambitieux de notre part d'essayer de répondre à des questions qui jusqu'ici sont restées sans réponse définitive. Nous pensons que chaque formalisme a sa particularité par rapport à sa capacité de représenter (et aussi de généraliser) des connaissances. Par conséquence, ce qu'on doit essayer de faire c'est de mieux connaître chaque type de méthode de représentation afin de pouvoir l'exploiter de la meilleure façon.

Dans cet esprit, nous avons proposé un stage de D.E.A. afin de mieux étudier 'la représentation et l'apprentissage de règles de haut niveau'. Cette étude a été développée par Gerardo Reyes-Salgado dans le cadre de son stage de D.E.A. [REY 97] réalisé sous la supervision de F. Osório au Laboratoire LEIBNIZ. Nous avons cherché à étudier des règles d'un niveau supérieur aux règles d'ordre 0 ou 0^+ traditionnellement employées avec les systèmes hybrides neuro-symboliques. Nous avons approfondi nos études sur les règles contextuelles et sur les règles avec des relations d'ordre supérieur entre leurs variables, c'est-à-dire, des règles où la relation entre deux variables est plus forte qu'une simple relation booléenne du type 'Et/Ou' (e.g. plus A est grand par rapport à B, plus il va influencer la prise d'une décision X). Nous allons décrire plus en détail ces recherches dans la section qui traite des résultats des applications d'INSS dans le problème de la balance (voir Section 5.3).

Les recherches que nous avons menées sur les règles de haut niveau ont été fructueuses [REY 97a] Elles nous ont en particulier permis de mieux nous situer par rapport aux problèmes de représentation de connaissances des systèmes hybrides neuro-symboliques. Rappelons que le système KBANN permet d'insérer et d'extraire seulement des règles d'ordre 0 (les entrées continues et les règles d'ordre 0^+ sont traitées en dehors du réseau par des techniques de pré-traitement) et le système SYNHESYS permet d'insérer et d'extraire seulement des règles simples d'ordre 0^+ (les variables sont considérées indépendamment, comme aussi dans les arbres de décision). Ces deux systèmes ne permettent pas de traiter des relations entre deux variables continues, comme par exemple : 'plus_grand_que' ou 'plus_petit_que'. Dans INSS, nous avons proposé des méthodes d'insertion de règles d'ordre 0^+ et des règles de haut niveau (voir Section 4.2.1) et nous avons aussi réalisé des études afin d'étendre le pouvoir de représentation de connaissances en utilisant des réseaux d'ordre supérieur, comme par exemple, les réseaux *Sigma-Pi* [REY 97, REY 97a].

Le point faible de notre système qui reste à améliorer est relatif aux type de règles que nous pouvons obtenir à partir de la partie connexionniste. Actuellement, nous pouvons extraire seulement des règles d'ordre 0, puisque nous nous sommes basés sur les méthodes d'extraction de règles utilisées dans KBANN (SUBSET et NofM). Cela représente une limitation non négligeable de notre système, car les règles extraites n'ont pas le même pouvoir de représentation de connaissances qu'au niveau de l'insertion de connaissances dans les réseaux. C'est pourquoi nous avons orienté nos recherches dans la direction de l'extraction de règles de haut niveau. Nous sommes en train de chercher des solutions à ce problème (G. Reyes a débuté une thèse cette année qui devra aborder ce sujet). Etant donné que les méthodes d'insertion de règles d'ordre 0^+ sont déjà opérationnelles, nous pensons que cela doit nous aider dans la recherche des méthodes capables d'extraire des règles plus complexes.

4.5 VERIFICATION ET VALIDATION DE CONNAISSANCES

Le module Valid permet de vérifier et de valider des règles symboliques et des données d'apprentissage. Nos méthodes de vérification et de validation ont été développées à partir de la nécessité de mettre en relation les règles et les exemples afin de trouver des inconsistances entre ces deux ensembles de connaissances. Ces algorithmes ont été réalisés et testés par Pierre-Yves Blond dans le cadre de son stage de D.E.A. [BLO 96] en utilisant le langage/moteur d'inférence CLIPS [GIA 93]. Les travaux développés par P.-Y. Blond, dans son stage de D.E.A., ont été réalisés sous la supervision de F. Osório.

Nous avons commencé nos recherches sur ce sujet à la suite des résultats obtenus dans l'extraction de règles à partir des réseaux. Comme nous l'avons déjà présenté dans la section précédente, les règles extraites peuvent contenir des information redondantes ou incohérentes, d'où l'idée d'appliquer des méthodes de vérification et validation des connaissances. De plus, nous avons choisi comme principe de base de notre système hybride, l'idée qu'il doit être conçu pour travailler avec des connaissances qui sont usuellement incomplètes et imparfaites. Or, si les connaissances ne sont ni complètes, ni parfaites, cela veut dire que ni les bases de règles, ni les bases d'exemples, ne doivent être considérées comme des connaissances absolues. Par conséquence, nous allons contester le mode de fonctionnement adopté par les réseaux KBANN, où les connaissances théoriques sont modifiées par apprentissage, mais où les connaissances empiriques (base d'exemples) ne sont jamais mises en question du point de vue de leur qualité. Nous avons choisi d'utiliser une approche qui est plutôt similaire à celle employée par le système SYNHESYS (acquisition constructive et interactive des connaissances). Dans ce cas de figure, un module comme Valid devient un outil très important qui permet l'implémentation de notre système hybride d'acquisition constructive de connaissances.

Avant de passer à la description du fonctionnement du module Valid, il faut définir de façon plus précise ce que l'on entend par vérification et validation de connaissances [FU 94]. La *vérification de connaissances* consiste à tenter de débusquer des erreurs structurales dans une base de règles (ou d'exemples), telles que : redondance, conflit, circularité, impasses, buts que l'on ne peut pas atteindre, attributs inutiles, attributs incohérents. Nous avons concentré nos efforts sur la vérification du type statique (cohérence statique) [FU 94, AYE 90], où nous cherchons à étudier l'adéquation entre les éléments des bases de connaissances et un modèle conceptuel du domaine défini à l'avance. Un exemple de vérification que l'on peut faire est le test de validité des valeurs d'entrée par rapport au type de la variable représentée : âge = valeur entière entre 0 et 130. Le langage CLIPS nous offre la possibilité d'établir ce type de contrôle d'une façon automatique.

D'un autre côté, la *validation de connaissances* porte sur la qualité des connaissances, ce qui va affecter la performance du système en termes de correction, d'efficacité, de robustesse, de sensibilité et d'adaptabilité. Nous nous sommes concentrés sur les aspects concernant les incohérences entre les deux bases de connaissances : la base de connaissances théoriques et la base de connaissances empiriques. Pour valider les connaissances, nous allons consulter la base de règles par inférence symbolique et comparer les réponses obtenues par rapport aux exemples contenus dans la base d'apprentissage. Dans le cas où les réponses sont contradictoires, nous faisons une analyse des données disponibles (activation par chaînage avant et arrière), afin de fournir à l'utilisateur le plus d'information possible qui puisse l'aider à identifier la cause de ce comportement anormal. Un exemple de validation que l'on peut faire est le cas où l'activation par chaînage avant avec un des exemples de la base d'apprentissage produit une réponse positive (sortie activée) et ce même exemple possède comme sortie désirée une valeur de réponse négative (sortie inactivée). Les statistiques sur le nombre de fois qu'un certain comportement anormal se produit, ainsi que l'indication précise des règles et des exemples liés aux cas problème, vont permettre à l'expert de décider de l'action à prendre (e.g.: donner priorité aux connaissances théoriques et ignorer l'exemple en l'effaçant de la base, corriger la base de règles selon les indications du système, obtenir plus d'exemples pratiques apparentés au cas problème, identifier une limitation imposée par rapport à la façon dont le problème a été posé, adapter le réseau par apprentissage des exemples et extraire des nouvelles règles pour remplacer les anciennes).

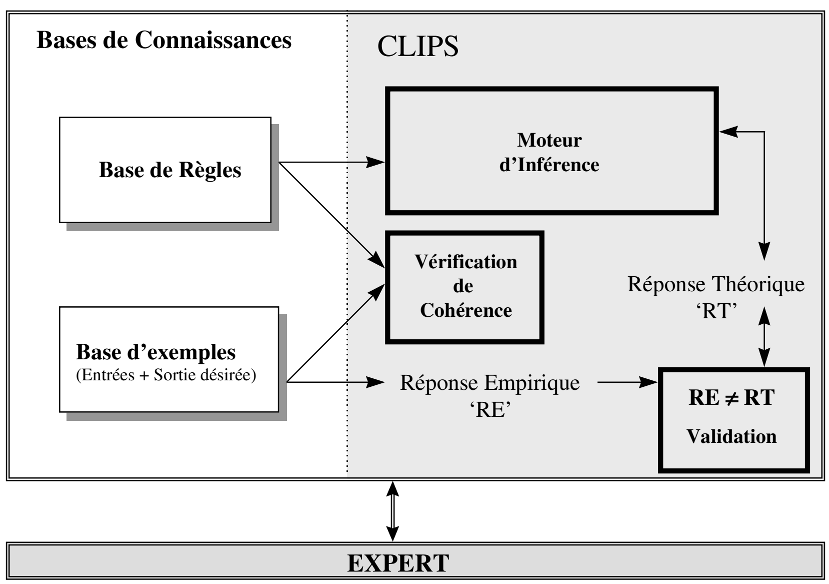

Figure 56 - Processus de vérification et de validation de connaissances

Le processus de vérification et validation de connaissances, tel qu'il a été étudié dans INSS (voir Figure 56), suit les étapes suivantes :

1. Vérification de la cohérence des règles et des exemples par une analyse du type statique. Dans cette phase, on va charger les règles et les exemples dans la mémoire de travail, en effectuant des vérifications sur leurs attributs. Le langage CLIPS permet de spécifier et de gérer des erreurs d'exception causées par l'utilisation de valeurs incorrectes dans les exemples. Nous pouvons aussi détecter facilement les règles qui ne sont jamais activées et les redondances.

2. Pour chaque exemple dans la base d'apprentissage (base de données constituée par les exemples avec leurs valeurs de sortie désirées), les règles sont activées par chaînage avant. Nous obtenons ainsi la 'réponse théorique', basée sur l'ensemble des règles disponibles. La valeur de la sortie désirée de l'exemple est appelée 'réponse empirique'. Nous avons quatre cas de figure possibles selon les valeurs de la 'réponse théorique' (sortie calculée - Sc) et de la 'réponse empirique' (sortie désirée - Sd)* :

   I. Sortie calculée active et correcte ('hit') : Sortie Sc=Sd=1.0, passer à l'exemple suivant;

   II. Sortie calculée inactive et correcte ('reject') : Sortie Sc=Sd=0.0, passer au suivant;

   III. Sortie calculée active et fausse ('false-alarm' - Faux_0): Sortie incorrecte, Sc≠Sd=0.0;

   IV. Sortie calculée inactive et fausse ('omission' - Faux_1): Sortie incorrecte, Sc≠Sd=1.0;

   * *Il faut retenir ici que les <u>sorties</u> dans INSS sont toujours <u>binaires</u> avec des valeurs approximées à 0.0 (inactivée) ou 1.0 (activée).*

3. Si la 'réponse théorique' n'est pas en accord avec la 'réponse empirique' (cas III et IV), le cas en question est signalé à l'expert et ensuite analysé plus en détails. Cette analyse consiste à faire une activation des règles par chaînage arrière et à établir à chaque pas en arrière l'importance des antécédents d'après l'état du conséquent. Les règles et ses antécédents sont ainsi classés en trois groupes : activation nécessaire ('want'), activation indéfinie ('nnns') et activation interdite ('forbid'). Le tableau ci-dessous indique la propagation en arrière de l'état des règles vers ses antécédents :

Etat d'activation de la Règle	Antécédents de la Règle	
	Opérateur du type 'OU'	Opérateur du type 'ET'
Want	NNNS	Want
NNNS	NNNS	NNNS
Forbid	Forbid	NNNS

Les règles et leurs antécédents qui ont des états d'activation incompatibles avec les états 'want' et 'forbid' établis par le chaînage arrière sont indiquées à l'expert/utilisateur.

4. Nous mettons à jour les statistiques de : nombre de 'hits', nombre de 'rejects', nombre de 'false-alarms' et nombre de 'omissions', ainsi que la mesure de l'utilisation ('utility') et du degré de correction ('correctness') relative des règles. La valeur 'utility' indique le nombre de fois qu'une règle à été déclenchée par rapport au nombre total d'exemples. La valeur 'correctness' indique le nombre de fois que la règle à été déclenchée et a conduit à une bonne réponse (cas I et II), par rapport au nombre total d'exemples.
5. Ensuite, le cas suivant de la base d'exemples est analysé (retour à l'étape '2').

Après avoir analysé tous les cas de la base d'exemples, nous présentons un rapport sur chaque exemple et sur chaque règle avec les totaux généraux cumulés dans l'étape '4' décrite ci-dessus (Pour obtenir une description plus détaillée des résultats affichés à la sortie du module Valid, voir le rapport [BLO 96]).

4.5.1 Discussion sur le Module de Vérification et Validation de Connaissances

Le module Valid reste le module le moins développé du système INSS. Cependant les fonctions de base actuellement disponibles se sont déjà montrées utiles en plusieurs occasions (voir le Chapitre 5). Le module Valid a été réalisé en utilisant le langage CLIPS. D'un point de vue pratique il n'est pas encore bien intègré à notre système (en particulier au module connexionniste). Nous avons décidé de mettre à disposition des utilisateurs certaines des fonctionnalités de ce module directement sur le module connexionniste (NeuSim), ce qui a simplifié la façon d'utiliser les fonctions basiques de validation de connaissances. Le module NeuSim nous offre lui aussi la possibilité de :

- séparer les exemples bien classés par rapport à une base de règles disponible, de ceux qui ont une réponse incompatible avec celle donnée par l'ensemble de règles (Pour ce faire, il suffit de coder une base de règles sur un réseau et de configurer le simulateur afin d'enregistrer les exemples incorrects dans un fichier à part);

- indiquer le nombre d'erreurs commises par un réseau étant donné une base d'exemples, tout en faisant la séparation des cas de 'faux_0' et de 'faux_1' (les réponses sont groupées dans 4 classes : correctes, incorrectes par 'faux_0', incorrectes par 'faux_1', incorrectes par indécision);

- indiquer la séquence de règles qui sont activées suite à la présentation d'un exemple particulier (règles compilées sous la forme d'un réseau), ce qui nous permettra de mieux analyser les résultats obtenus par l'activation du réseau.

D'un autre côté, le module Extract lui aussi offre des possibilités de simplification des règles extraites à partir d'un réseau, à travers une vérification des règles obtenues et l'élimination des règles inutiles. Ce module nous permet aussi d'identifier des attributs d'entrée qui n'ont pas d'influence sur les résultats obtenus à la sortie.

Toutes ces fonctions de vérification et de validation de connaissances nous ont permis d'obtenir des résultats intéressants dans deux cas d'application du système INSS. Dans une application d'aide au diagnostic médical des comas toxiques (voir section 5.2), nous avons trouvé des anomalies d'apprentissage, alors même que le réseau étudié avait un bon taux de réussite. Une analyse plus détaillée des résultats (nombre de 'faux_0' et 'faux_1') nous a permis d'identifier que le réseau donnait une réponse unique dans sa sortie, avec des valeurs toujours égales à 0 (absence d'une substance toxique X) ou toujours égales à 1 (présence d'une substance toxique X), selon le type d'exemples dominants dans la base d'apprentissage. Dans une autre application, le problème de la balance (voir Section 5.3), nous avons pu identifier des problèmes liés à l'incapacité du réseau de bien représenter de façon interne une règle de haut niveau [REY 96]. Une étude des 'cas problème' nous a permis de mieux comprendre pourquoi nous avions des problèmes pour apprendre tous les exemples. Nous allons aborder ces deux cas d'application du système INSS dans le chapitre suivant.

Nous pensons que le module de vérification et de validation de connaissances doit être encore mieux étudié, car c'est justement à partir des 'conflits cognitifs' (e.g.: contradictions et erreurs commis) que nous faisons évoluer nos connaissances, d'où la grande importance de cette étape dans un processus global d'acquisition constructive de connaissances. Le grand nombre de recherches en cours sur la vérification et la validation des bases de connaissances [CRA 97, NAZ 97, FU 94, AYE 90] montre bien l'importance de ce domaine qui reste ouvert à la recherche.

Finalement, soulignons que les outils de vérification et de validation que nous avons mis à disposition des utilisateurs dans le système INSS ne représentent qu'une partie des méthodes que l'on aimerait avoir a sa disposition. Nous laissons à des études futures certains aspects tels que : la détection de boucles dans les règles, la possibilité d'associer un facteur de certitude aux règles et aux exemples, la détection de contradictions internes à la base d'exemples ou de règles, la validation de règles avec des antécédents non-discrets (validation des intervalles d'activation pour les valeurs continues).

4.6 CONSIDERATIONS FINALES SUR LE SYSTEME INSS ET L'ACQUISITION CONSTRUCTIVE DE CONNAISSANCES

Le système INSS a été conçu afin d'utiliser toutes les connaissances disponibles sur un problème: théoriques et empiriques. Or, ces connaissances sont généralement incomplètes et peuvent contenir des incorrections, et en plus, elles peuvent évoluer avec le temps. En prenant ces idées comme nos concepts de base, nous avons construit notre système hybride neuro-symbolique d'acquisition constructive de connaissances.

Le système INSS s'est inspiré d'autres systèmes, comme KBANN et SYNHESYS, et il a apporté plusieurs améliorations par rapport à eux. De plus, la façon d'intégrer les connaissances théoriques et empiriques et de les faire évoluer, telle que nous l'avons réalisé dans INSS, à été aussi inspiré par ce que nous avons appelé de la "*méthodologie de l'élève-chercheur*" [OSO 95]. La méthodologie de l'élève-chercheur est une caricature du comportement d'un individu qui commence comme un élève mais qui pourra aussi devenir un jour un chercheur. Le côté *élève* représente la phase de l'acquisition de connaissances où il reçoit des connaissances théoriques, c'est-à-dire, des connaissances a priori concernant le domaine traité. Ces connaissances lui sont fournies par un maître (expert). Par exemple, un élève dans un cours de physique va commencer par l'apprentissage des lois (règles) physiques qui représentent ce domaine de connaissances.

L'élève doit ensuite procéder à des expériences pratiques pour valider et améliorer ses connaissances, et peut être ainsi il va devenir un *chercheur*. Alors, le chercheur est symbolisé par une personne qui essaye d'utiliser ses connaissances théoriques initiales et ses expériences pratiques afin d'arriver à des nouvelles connaissances théoriques sur le domaine de recherche. Une fois que les nouvelles connaissances proposées par le chercheur deviennent 'acceptables' (validées), elles vont être ensuite transmises aux élèves ; et ainsi la boucle est bouclée. Notre "méthodologie de l'élève-chercheur" est donc un modèle assez générique de l'acquisition constructive de connaissances. Une approximation simpliste de ce modèle peut être retrouvé dans la façon de fonctionner de notre système INSS.

Nous sommes conscients que notre système ne représente qu'un modèle simplifié de cette méthodologie, puisque les connaissances sont exprimées par un langage très simple. Cependant, le système INSS présente déjà certaines propriétés très intéressantes, dues au modèle d'acquisition constructive de connaissances adopté, qui font de ce système un outil intéressant pour l'étude des processus cognitifs relatifs à l'apprentissage humain.

Comme nous l'avons vu tout au long de ce chapitre, le système INSS apporte toute une série d'améliorations à son prédécesseur KBANN. Les propriétés le plus intéressantes d'INSS sont donc :

- INSS a la possibilité d'insérer par compilation des règles d'ordre 0^+ (et des règles de haut niveau) dans un réseau;

- INSS offre la possibilité de travailler avec des données qualitatives aussi bien qu'avec des données quantitatives;

- INSS utilise un réseau constructif, ce que lui permet de faire évoluer les connaissances du réseau ainsi que son structure. L'algorithme d'apprentissage utilisé est reconnu par sa performance supérieure à l'algorithme classique de la Rétro-propagation;

- INSS utilise une méthode d'extraction de règles incrémentale. Le processus d'extraction de règles que nous avons réalisé est plus performant que celui de son prédécesseur KBANN, car il permet d'extraire seulement les nouvelles connaissances acquises, tout en préservant les connaissances préalablement introduites dans le réseau;

- INSS permet de simplifier le réseau (couper connexions et éliminer unités) avant l'extraction de règles. Nous essayons d'extraire les règles les plus représentatives des connaissances codées dans le réseau;

- INSS offre la possibilité de valider les nouvelles connaissances acquises par rapport aux anciennes;

- INSS est un outil qui peut être facilement adaptable à des applications dans différentes domaines.

Toutefois, il reste encore des problèmes à résoudre et des recherches à réaliser afin de perfectionner encore plus cet outil. Deux points principaux à développer sont :

- le développement de nouvelles méthodes d'extraction de règles afin de pouvoir extraire des règles de plus haut niveau à partir des réseaux et de mieux exploiter les connaissances qui y sont représentées;

- le perfectionnement des outils de vérification et de validation de règles disponibles.

Il y a d'autres perspectives d'étude sur le système INSS, tels les deux sujets que l'on vient de décrire ci-dessus, mais aussi :

- la possibilité d'utiliser un moteur d'inférence plus puissant à la place du langage CLIPS, comme pour exemple l'ECLIPSE (une version améliorée de CLIPS qui inclut la possibilité de réaliser un chaînage arrière);

- l'étude des possibilités d'adaptation des intervalles définis pour les variables d'entrée continues. Nous avons réalisé des fonctions qui permettent de coder des règles d'ordre 0^+ sur un réseau, mais nous n'avons pas approfondi les possibilités d'adaptation de ces règles vers un niveau plus fin que celui de l'ajout de nouvelles unités au réseau;

- l'insertion par compilation de fonctions plus complexes dans les réseaux. Nous avons envisagé la possibilité d'ajouter des unités qui auraient des liens (interface) vers des fonctions en 'C'. L'activation d'une unité pourra déclencher l'exécution d'une fonction complexe qui serait codée dans le simulateur du réseau. Cette approche nous permettrait d'avoir des unités spécialisées dans des tâches complexes et très spécifiques;

- l'incorporation du temps dans les réseaux. INSS ne peut traiter le temps qu'en changeant la façon de coder les données (fenêtre temporelle - contexte). On peut envisager le développement d'un système hybride comme INSS basé sur des réseaux récurrents.

Pour compléter, l'application d'une *méthode d'apprentissage actif* pourrait être envisagé dans le cadre du système INSS. Par apprentissage actif, nous entendons la capacité d'un système d'apprentissage automatique d'interagir avec l'environnement (expert, sources de données). Par contre, un système va être considéré comme passif quand il n'a pas de réactions (Il accepte tout) face à des informations qui lui arrivent. Par exemple, le système INSS pourrait demander automatiquement à l'expert d'avantage d'exemples autour de certaines connaissances mal apprises. Le système peut ainsi faciliter et améliorer la tâche d'acquisition de ses connaissances.

Dans un cadre plus général, à propos des systèmes hybrides, nous pensons qu'une voie intéressante de recherche est l'étude d'un système qui puisse combiner différentes approches dans un seul outil. L'approche utilisée par SCANDAL pourrait servir à combiner des systèmes tel que INSS (réseaux de type PMC), SYNHESYS (réseaux à prototypes) et ProBis (CBR+ANN, avec la possibilité de traiter les cas atypiques).

Finalement, un des points très importants de l'étude du système INSS a été son application à des problèmes pratiques pour mieux connaître ses propriétés ainsi que ses limitations. Nous avons testé le système sur différents types d'applications que nous allons présenter dans le chapitre suivant : les problèmes du Moine, l'aide au diagnostic médical, le problème de la balance et la robotique autonome.