Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Algorithmes génétiques et leurs applications aux réseaux de neurones

Piazza, Salvino

Award date:
2003

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

FUNDP
Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique

Année académique 2002-2003

ALGORITHMES GÉNÉTIQUES ET LEURS APPLICATIONS AUX RÉSEAUX DE NEURONES

Ir. Salvino Piazza

Mémoire présenté en vue de l'obtention
du grade de Licencié en Informatique

Résumé

GA, Algorithmes génétiques, NN, Réseau de neurones, GANN, Evolution

Le but de ce travail de fin d'étude est de décrire les applications des algorithmes génétiques aux réseaux de neurones. Il se compose de deux grandes parties. La première partie, qui est la plus importante, est une compilation d'articles sur ce sujet. Elle commence par une introduction aux algorithmes génétiques et aux réseaux de neurones, ensuite elle montre de quelle manière ces deux grands domaines ont été intégrés. Ce travail décrit 5 grands types d'application : L'apprentissage par algorithme génétique des poids de réseau de neurones (Méthodes classique et hybride), les algorithmes génétiques utilisés pour trouver l'architecture optimale d'un réseau de neurones (Méthodes d'encodage direct, d'encodage paramétrique, d'encodage de grammaire et de programmation génétique), les algorithmes génétiques pour trouver de bonnes règles d'apprentissage de réseaux de neurones, les algorithmes génétiques pour diminuer le nombre d'entrées de réseau de neurones et les algorithmes génétiques utilisés pour sélectionner les éléments pertinents des groupes d'apprentissage. La seconde partie de ce travail, est de type programmation. Elle est constituée d'un programme permettant de tester certains des algorithmes exposés dans la première partie. Elle comprend une description du programme d'algorithmes génétiques appliqué aux réseaux de neurones et décrit les résultats obtenus avec ce programme, pour trois applications.

Abstract

GA, Genetic Algorithms, NN, Neural Networks, GANN, Evolutionary

The aim of this bachelor thesis is to review different genetic algorithm (GA) combinations for neural networks (NN) applications. This thesis consists in two great parts. The first part, which is the most important and a compilation of different articles related to that subject, begins with an introduction on genetic algorithms and neural networks. Then it shows how those two great domains merged. This work described 5 great kinds of applications: using GA to find out NN connection weights (Classical and Hybrid methods); using GA to find out NN architecture (Direct encoding, parametric encoding, grammar encoding and genetic programming); using GA to find out NN learning rules; using GA to decrease the number of NN inputs and using GA to select good elements of learning groups of NN. The second part of this work is the implementation of a program. It has been used for testing some applications described in the first part of the works. A description of the program is given in the book. Program results for three different applications are also included.

Avant-propos

Je voudrais en premier lieu exprimer tous mes remerciements à Jean-Paul Leclercq, Professeur à l'Institut d'informatique des Facultés Universitaires Notre-Dame de la Paix, de Namur, pour m'avoir donné la possibilité d'effectuer ce travail de fin d'étude dans un domaine qui m'a toujours passionné, l'optimisation. Je le remercie également pour les lectures attentives qu'il a faites de ce travail aux différentes étapes de sa réalisation.

Mes remerciements vont également à tous ceux qui ont, de près ou de loin, participé à l'élaboration de ce document, et à tous ceux qui par un simple mot ou geste, m'ont aidé et encouragé tout au long de mes études, je pense à mes parents, à mes beaux-parents, à ma sœur Rita, à mon parrain, à ma marraine, à ma cousine Laura, mon cousin Salvatore, à tous mes amis d'enfance ainsi qu'aux petits frères, à Patrick, à Dominique et bien sûr à mon épouse Julie.

Je tiens à remercier tout particulièrement Micheline Laudelout et "Parrain" (Professeur Henry Laudelout) pour tous les livres d'informatique qu'ils m'ont prêtés. Ils m'ont été d'une aide précieuse durant ces deux dernières années d'études et m'ont permis d'aller plus loin. Que ces remerciements soient grands et qu'ils arrivent tout là haut près de lui.

J'aurai à présent une pensée particulière pour mon épouse Julie, et une fois de plus je la remercie pour tout. Je lui dédie ce travail de fin d'étude.

Enfin, je remercie les membres du jury pour le temps qu'ils vont passer à la lecture de ce travail, qu'ils trouvent ici l'expression de ma gratitude.

Tables des matières

1 INTRODUCTION 1

2 CHAPITRE 1 : ALGORITHMES GÉNÉTIQUES 2

2.1 INTRODUCTION 2

2.2 PRINCIPES FONDAMENTAUX DES ALGORITHMES GÉNÉTIQUES 3

2.2.1 Définitions 3

2.2.2 Les différentes étapes 3

2.3 MÉTHODES AVANCÉES 5

2.3.1 Codage non binaire 5

2.3.2 Elitisme 6

2.3.3 Le scaling 6

2.3.4 Le Sharing 8

2.3.5 Le Sharing clusterisé 8

2.3.6 Le recuit simulé (Simulated Annealing) 10

2.3.7 Mutation adaptative 12

3 CHAPITRE 2 : RÉSEAUX DE NEURONES 13

3.1 HISTORIQUE 13

3.2 LE RÉSEAU DE NEURONES BIOLOGIQUES 14

3.3 LE RÉSEAU DE NEURONES ARTIFICIELS 15

3.3.1 Les neurones 16

3.3.2 Les connexions entre unités 16

3.3.3 Activation et règle de sortie 17

3.4 TOPOLOGIE DES RÉSEAUX 17

3.5 APPRENTISSAGE DES RÉSEAUX DE NEURONES 18

3.6 RÉSEAUX DE NEURONES À UNE SEULE COUCHE 18

3.6.1 Le Perceptron 18

3.6.2 Règle d'apprentissage du Perceptron 19

3.6.3 L'Adaline (Adaptive linear element) 20

3.6.4 Le problème du XOR (ou exclusif) 20

3.7 RÉSEAUX MULTICOUCHES FEED-FORWARD 21

3.7.1 La règle des deltas généralisée (Backpropagation BP du gradient) 21

3.7.2 Taux d'apprentissage γ et Momentum α 23

3.7.3 Apprentissage par succession d'individus uniques 24

3.7.4 Déficience de la méthode de back-propagation 24

3.7.5 Algorithmes avancés 25

3.7.6 Les effets du nombre d'individus de l'ensemble d'apprentissage 25

3.7.7 Les effets du nombre de couches cachées 26

3.8 LES RÉSEAUX RÉCURRENTS 27

3.8.1 Généralité 27

3.8.2 Le réseau de Hopfield 27

3.8.3 Le réseau de Kohonen 31

4 CHAPITRE 3 : APPLICATIONS DES ALGORITHMES GÉNÉTIQUES AUX RÉSEAUX DE NEURONES 33

4.1 INTRODUCTION 33

4.2 APPRENTISSAGE DES POIDS D'UN RÉSEAU DE NEURONES PAR ALGORITHME GÉNÉTIQUE GA 35

4.2.1 Introduction 35

4.2.2 L'encodage 35

4.2.3 Ordre de concaténation 36

4.2.4 Le problème de la permutation (Competing Conventions Problem) 37

4.2.5 La fonction d'évaluation 38

4.2.6 Apprentissage des poids par algorithme génétique GA 38

4.2.7 Apprentissage hybride (algorithme génétique GA + rétropropagation du gradient BP) 44

4.2.8 Comparaison de l'apprentissage par GA, BP et hybride (BP + GA) 46

4.2.9 Conclusion 47

4.3 ARCHITECTURE OPTIMALE DE RÉSEAUX DE NEURONES PAR ALGORITHMES GÉNÉTIQUES 48

4.3.1 Introduction 48

4.3.2 Encodage direct 49

4.3.3 Encodage paramétrique 51

4.3.4 Encodage de grammaire 52

4.3.5 Encodage par programmation génétique 65

4.4 RECHERCHE DE RÈGLES D'APPRENTISSAGE DE RÉSEAU DE NEURONES PAR ALGORITHMES GÉNÉTIQUES 71

4.4.1 Optimisation de règles d'apprentissage: méthode de Chalmers 71

4.4.2 Optimisation de taux d'apprentissage γ et du momentum α pour la rétropropagation du gradient BP 75

4.4.3 Autres applications 76

4.5 ALGORITHMES GÉNÉTIQUES POUR LA SÉLECTION DES ENTRÉES LORS D'UNE PHASE D'APPRENTISSAGE DE RÉSEAU DE NEURONES 77

4.5.1 Sélections des neurones d'entrées d'un réseau de neurones 77

4.6 SÉLECTIONS DES ÉLÉMENTS DU GROUPE D'APPRENTISSAGE DE RÉSEAU DE NEURONES 79

5 CHAPITRE 4 : PROGRAMMES 81

5.1 PACKAGE GA 81

5.1.1 Package GA.Individu 81

5.1.2 Package GA.Population 83

5.2 PACKAGE NN 85

5.2.1 Package NN. Struct_In 85

5.2.2 Package NN.Struct_Out 86

5.2.3 Package NN.Pattern 86

5.2.4 Package NN.Struct 87

5.2.5 Package NN.FeedForward 87

5.2.6 Package NN.BackPropagation 87

5.3 PACKAGE GA_NN 88

5.3.1 Package GA_NN.Opt_Struct 88

5.3.2 Package GA_NN.Opt_Weight 88

5.4 PACKAGE TOOLS 89

5.4.1 Package Tools.Tri 89

5.4.2 Package Tools.Gen_Aleat 89

5.5 PACKAGE TEST 89

5.5.1 Package Test.NN_BackPropagation 89

5.5.2 Package Test.Ga_NN_Opt_Learn 89

5.5.3 Package Test.Ga_NN_Opt_Weight 89

5.5.4 Package Test.Ga_NN_Opt_Struct 89

5.6 INTERACTION DES DIFFÉRENTES CLASS GA ET NN 90

5.6.1 Configuration des différents modules optimisation des paramètres "Taux d'apprentissage" et "momentum" 90

5.6.2 Configuration des différents modules optimisation des poids d'un réseau de neurones par GA 91

5.6.3 Configuration des différents modules optimisation de structure de réseau de neurones par GA (méthode de Miller) 92

6 CHAPITRE 5 : APPLICATIONS 93

6.1 OPTIMISATION DU TAUX D'APPRENTISSAGE γ ET DU MOMENTUM α POUR LA RÉTROPROPAGATION DU GRADIENT BP 93

6.1.1 Algorithme 93

6.1.2 Description de l'application 93

6.1.3 Les paramètres utilisés 93

6.1.4 Résultats 94

6.2 COMPARAISON DE DIFFÉRENTES MÉTHODES D'APPRENTISSAGE DES POIDS DE RÉSEAUX DE NEURONES 95
6.2.1 Algorithme *95*
6.2.2 Description de l'application *95*
6.2.3 Les paramètres utilisés *95*
6.2.4 Résultats *96*
6.3 OPTIMISATION DE STRUCTURE DE RÉSEAU DE NEURONES: MÉTHODE DE MILLER 97
6.3.1 Algorithme *97*
6.3.2 Description de l'application *98*
6.3.3 Les paramètres utilisés *98*
6.3.4 Résultats *99*
7 CONCLUSION 100
8 BIBLIOGRAPHIE 101
9 ANNEXES 107
9.1 LISTING DES DIFFÉRENTS PROGRAMMES JAVA 107
9.1.1 Individu *107*
9.1.2 Population *123*
9.1.3 NN *146*
9.1.4 GA_NN *171*
9.1.5 Tools *177*
9.1.6 Test *179*

1 Introduction

Les applications des algorithmes génétiques sont nombreuses et variées. Ce travail de fin d'étude s'intéresse plus particulièrement aux applications liées aux réseaux de neurones. Il se compose de deux grandes parties.

La première partie, qui est la plus importante, est une compilation d'articles dédiés à ce sujet. Cette partie s'étend sur les trois premiers chapitres. Les chapitres 1 et 2 présentent les différents éléments de bases nécessaires à la compréhension des algorithmes génétiques et des réseaux de neurones. Le chapitre 3 décrit en détail de quelle manière ces deux grands domaines ont pu être intégrés. Ce chapitre contient 5 grands types d'applications différentes:

- L'apprentissage par algorithme génétique des poids de réseau de neurones: Méthode classique et méthode hybride.
- Les algorithmes génétiques pour trouver l'architecture optimale d'un réseau de neurones : Méthode d'encodage direct, d'encodage paramétrique, d'encodage de grammaire et de programmation génétique.
- Les algorithmes génétiques pour trouver de bonnes règles d'apprentissage de réseaux de neurones.
- Les algorithmes génétiques pour diminuer le nombre d'entrées de réseau de neurones, afin de garder uniquement les entrées utiles.
- Les algorithmes génétiques utilisés pour sélectionner les éléments pertinents des groupes d'apprentissage.

La seconde partie de ce travail est de type programmation. Elle a pour but la réalisation d'un programme permettant de tester certains des algorithmes exposés dans la première partie. Elle s'étend sur les chapitres 4 et 5. Le chapitre 4 décrit le programme d'algorithmes génétiques appliqué aux réseaux de neurones. Le chapitre 5 décrit les résultats obtenus avec ce programme, pour trois applications particulières: L'optimisation des paramètres de l'algorithme de rétropropagation du gradient, l'optimisation de poids de réseaux de neurones et l'optimisation d'architecture de réseaux de neurones.

2 Chapitre 1 : Algorithmes génétiques

2.1 Introduction

Les algorithmes génétiques sont basés sur un processus biologique naturel : l'évolution des espèces vivantes. Cette évolution se base sur deux mécanismes : La sélection naturelle et la reproduction. La sélection fait que seuls les individus les plus aptes survivent. La reproduction permet d'obtenir de nouveaux individus aux potentialités nouvelles.

Les premiers travaux sur les algorithmes génétiques remontent aux années cinquante et sont le fruit des recherches de plusieurs biologistes américains. Ces travaux ont permis de simuler des structures biologiques sur ordinateurs.

Entre 1960 et 1970, J. Holland développa sur base de ces travaux précédents, les premiers principes fondamentaux des algorithmes génétiques ([Hol62] et [Hol75]). Il orienta ainsi cette technique au domaine de l'optimisation mathématique. Cependant la faible puissance des ordinateurs de l'époque ne permit pas d'exploiter ces algorithmes sur des problèmes de grande taille.

Fin des années 1980, D. Goldberg décrit dans son ouvrage, l'utilisation d'algorithmes génétiques pour résoudre des problèmes concrets. Cet ouvrage est devenu la publication de référence [Gol89] et a permis de mieux faire connaître les algorithmes génétiques et de relancer l'intérêt pour cette technique.

Les démonstrations théoriques de convergence ne sont apparues que très tard, de par la complexité théorique induite par ces algorithmes. En 1993, une démonstration complète de convergence stochastique est établie par R. Cerf [Cer94]. Cependant les résultats théoriques sont difficilement exploitables en pratique. La convergence efficace fait encore beaucoup appel au savoir-faire de l'utilisateur.

Les algorithmes génétiques ont le grand avantage d'être robustes, rapides, ne demandant aucune connaissance sur le système à optimiser. Ce qui permet d'obtenir rapidement l'optimum de fonctions fortement non convexes, ce qui est un gros avantage par rapport aux autres algorithmes classiques d'optimisation.

2.2 Principes fondamentaux des algorithmes génétiques

2.2.1 Définitions

La Population : est définie comme un ensemble d'individus.

Individu : est constitué d'un ou plusieurs chromosomes.

Chromosome : est composé d'un nombre de gènes égal aux variables à optimiser et dont les valeurs sont les allèles.

Fitness: est une valeur permettant de mesurer si la valeur de la fonction est proche de la valeur à optimiser.

2.2.2 Les différentes étapes

Les algorithmes génétiques élémentaires utilisent le codage binaire. Ils se décomposent en 7 grandes étapes:

1) Codage

Chaque chromosome est représenté par une suite de 1 ou de 0, inspiré des acides aminés des chromosomes biologiques.

Figure 2-1 : Exemple de codage d'un individu avec 4 chromosomes composés de 3 locus (bits)

2) Initialisation

Tirage aléatoire des chromosomes des individus constituant la population initiale.

3) Evaluation

Calcul des fitness des individus de la population.

4) Sélection

Choix des meilleurs individus de la population, afin de créer une nouvelle population mieux adaptée. Il y a plusieurs grands types de sélection. La *roulette wheel selection,* les *tournois* et le *stochastic remainder without replacement selection* [Mic92].

Roulette wheel selection:

A chaque individu est associé un segment dont la longueur est proportionnelle à sa fitness. Ensuite ces segments sont mis bout à bout, un tirage aléatoire d'une valeur est effectué entre les deux bornes de cet axe. Les bons individus (grands segments) seront plus souvent sélectionnés que les petits. Cette sélection bien que très simple présente l'inconvénient, pour des petites populations, de n'avoir pas un tirage représentatif et par conséquent il y a un risque de perdre les bons éléments.

Tournois :

Deux individus sont choisis avec l'algorithme de la *Roulette wheel selection* et le meilleur est gardé. Première variante, cet algorithme peut être appliqué à plus de 2 individus. Seconde variante, l'algorithme de sélection peut être différent de la *Roulette wheel selection.*

Stochastic remainder without replacement selection.

Cette technique permet de conserver, à chaque sélection, les meilleurs individus en proportion de leur fitness. Cet algorithme comprend les étapes suivantes pour chaque individu i:

- Calcul du rapport de la fitness de l'individu i par la fitness moyenne de tous les individus.
- On prend ensuite la partie entière de ce rapport.
- Ce chiffre définit le nombre de fois que l'individu i sera inclus dans la nouvelle population.
- Si la population ne contient pas assez d'individus, elle est complétée en utilisant la *roulette wheel selection.*

5) Croisement (*Cross-over*)

Croisement aléatoire des individus choisis aléatoirement dans la population. La proportion d'éléments croisés varie généralement entre 30 et 60 %. Cette étape s'inspire du crossing-over des chromosomes biologiques. Ci-dessous un exemple de crossing-over simple à 1 point et un autre exemple de crossing-over à 2 points.

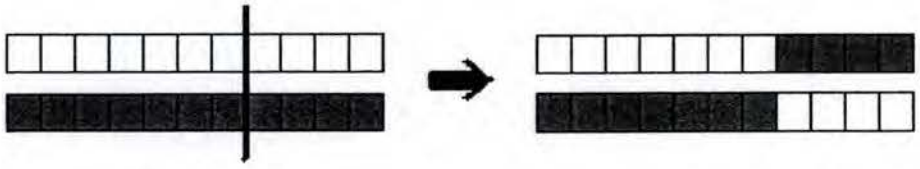


Figure 2-2 : Exemple de croisement "Crossing-over" simple à 1 point

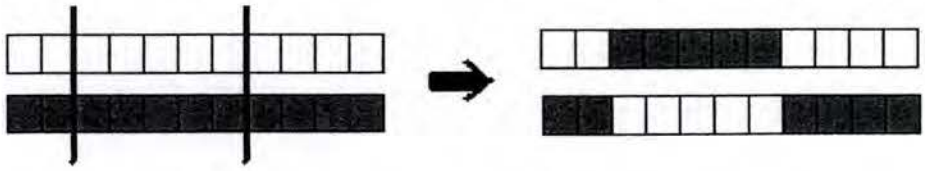


Figure 2-3 : Exemple de croisement "Crossing-over" à 2 points

6) Mutation

Modification aléatoire d'individus choisis aléatoirement dans la population. La proportion d'éléments mutés est souvent propre à l'application étudiée.

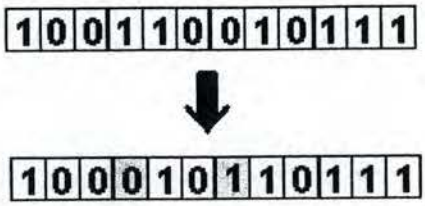


Figure 2-4 : Exemple de la mutation de 2 bits aléatoires

7) Terminaison

En fonction du critère de convergence, on recommence à l'étape 3 ou on termine.

2.3 Méthodes avancées

2.3.1 Codage non binaire

Il a été introduit par les chercheurs, pour remédier aux inconvénients du codage binaire.

Les inconvénients du codage binaire

- Deux nombres très proches ont parfois une distance de Hamming très élevée. Ce phénomène introduit des discontinuités dans l'espace de recherche. La distance de Hamming entre deux séquences de bits (de longueur égale), est le nombre de bits distincts entre ces deux séquences, la comparaison portant sur des bits de même position. Par exemple la distance de Hamming entre 31 (011111) et 32 (100000) est de 6.
- La mutation d'un bit a des conséquences totalement différentes si elle porte sur un des bits de poids fort (bits les plus à gauche dans la chaîne de bits) ou sur un des bits de poids faible (bits les plus à droite dans la chaîne de bits).

Le codage binaire classique a donc été remplacé avantageusement par des tableaux d'entiers ou beaucoup plus fréquemment par des tableaux de réels. La modification du codage a engendré une modification des étapes de croisement et de mutation.

Croisement

Le croisement de nombres réels se fait très fréquemment sur base du modèle barycentrique, avec α choisi aléatoirement entre -0.5 et 1.5.

$$X1' = \alpha \,.\, X1 + (1 - \alpha) \,.\, X2$$

$$X2' = \alpha \,.\, X2 + (1 - \alpha) \,.\, X1$$

Avec X1 et X2 les parents et X1' et X2' les fils.

Afin de ne pas trop s'éloigner des parents le croisement s'effectue sur une partie des chromosomes.

Mutation

Elle se fait par addition d'un bruit Gaussien à certains éléments de l'individu choisis aléatoirement.

2.3.2 Elitisme

Lors des croisements et des mutations, il se peut que le meilleur individu, l'élite, soit endommagé de façon irréversible. Ce problème peut également se produire pour certaines méthodes de sélection, comme par exemple la *roulette wheel selection* ou le *tournoi*. Le meilleur élément est ainsi perdu et ne sera peut-être jamais reconstruit. Afin d'éviter ce problème, la notion d'élitisme a été introduite. Ce qui nous permet de marquer le meilleur individu lors de chaque génération, afin de le conserver jusqu'à la génération suivante.

Afin d'améliorer encore l'algorithme, cette notion peut être étendue à plus d'un individu. On calcule le pourcentage du fitness des individus par rapport au fitness de l'élite. Ensuite, les individus ayant un pourcentage supérieur au "pourcentage d'élite", feront partie des individus élites. Tous ces individus seront donc sélectionnés pour la génération suivante sans être ni croisés, ni mutés. Ce pourcentage d'élite varie généralement entre 90 et 95 %.

2.3.3 Le scaling

Les processus de sélection présentés ci-avant sont très sensibles aux écarts de fitness. Dans certains cas, un individu trop bon risque d'être reproduit trop souvent et parfois d'éliminer certains individus, qui à terme auraient pu donner également des bons éléments.

Le *scaling* ainsi que le *sharing* (voir point suivant) empêchent les individus "forts" d'éliminer les "faibles".

Le *scaling* [Mic92] est une mise à l'échelle qui permet de réduire ou d'augmenter artificiellement l'écart de *fitness* entre les individus, la sélection n'opère plus sur le *fitness* mais sur ces *fitness* artificielles. Cette opération permet ainsi, par exemple, au début de favoriser l'exploration et en fin d'algorithme de privilégier la croissance des individus forts.

Parmi les fonctions de scaling, on a le scaling linéaire et le scaling exponentiel. F^r la fitness réelle avant scaling et F^a la fitness artificielle après scaling.

Le scaling linéaire

$$\mathbf{Fa = a.\ Fr + b}$$

$$a = \frac{max' - min'}{max - min} \qquad b = \frac{min' \,.\, max - min \,.\, max'}{max - min}$$

Ce scaling est statique par rapport au numéro de génération ce qui pénalise la fin de la convergence où l'on désire justement favoriser les modes dominants.

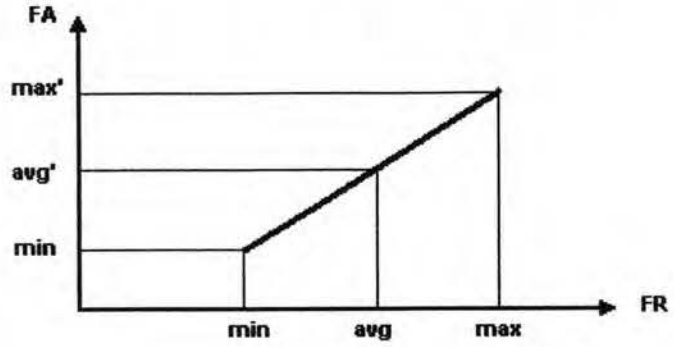


Figure 2-5 : La fonction de scaling linéaire

Le scaling exponentiel

$$\mathbf{Fa = Fr^k}$$

Pour **k <1**, (par exemple proche de 0), aucun individu n'est vraiment favorisé, seuls les individus de très faible fitness sont défavorisés. Cela permet une très bonne exploration de l'espace, ce qui est bénéfique en début de processus.

Pour **k=1**, scaling inopérant.

Pour **k>1**, les écarts des individus dominants sont exagérés vers les fortes valeurs, ce qui est bénéfique en fin de processus.

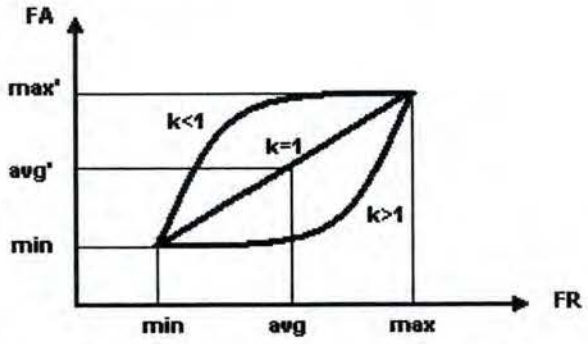


Figure 2-6 : La fonction de scaling exponentiel

On voit donc que faire varier k des faibles valeurs vers les fortes valeurs entre le début et la fin du processus d'évolution est très intéressant. Voici une fonction fréquemment utilisée à cet effet.

$$k = \left(\tan\left(\left(\frac{n}{N+1} \right) . \frac{\pi}{2} \right) \right)^{\rho}$$

Avec n le nombre de générations courantes et N le nombre total de générations. On prend généralement $\rho = 0.1$

2.3.4 Le Sharing

Cet algorithme a pour but de pénaliser la fitness des individus fortement agglomérés, les fitness vont être modifiées avant l'étape de sélection. L'objectif du sharing ([GR87]) est de repartir sur chaque sommet de la fonction à optimiser, un nombre d'individus proportionnel à la fitness associée à ce sommet. Le sharing a donc pour but d'éviter le rassemblement de beaucoup d'individus autour d'un même sommet. On va donc introduire la notion de distance, cette distance est utilisée pour calculer le nouveau sharing de la manière suivante.

$$Fi' = \frac{Fi}{mi'}; mi' = \sum_{j=1}^{j=N} S(d(Xi, Xj))$$

$$S(d) = 1 - \left(\frac{d}{\sigma_{Share}}\right)^{\alpha} \quad si \quad d < \sigma_{Share}$$

$$S(d) = 0 \quad si \quad d \geq \sigma_{Share}$$

σ_{share} délimite le voisinage d'un point, α est un autre paramètre (voir figure ci-dessous).

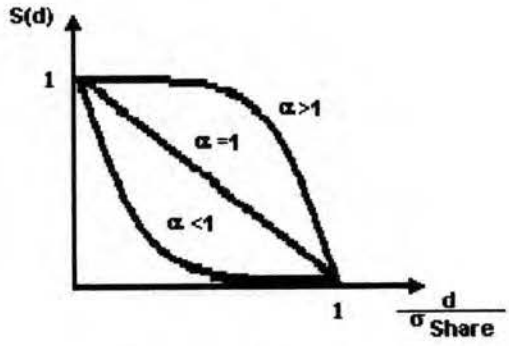


Figure 2-7 : Fonction S(d)

Le sharing est très efficace mais demande N^2 calculs de distance entre les chromosomes, or la complexité des algorithmes génétiques est en N sans sharing. Passer à N^2 est donc très pénalisant pour des populations avec beaucoup d'individus. Pour solutionner ce problème on passe au sharing clusterisé.

2.3.5 Le Sharing clusterisé

Comme pour le Sharing classique, cet algorithme a pour but de pénaliser la fitness des individus fortement agglomérés. Nous allons donc calculer à nouveau des distances entre individus. La grande différence ici, réside dans le fait que l'on ne va pas calculer la distance par rapport à tous les autres individus de la population, mais la distance entre les individus appartenant à un même groupe. Ces groupes d'individus seront appelés des clusters [ChSe95].

Pour répartir ces individus dans les différents clusters, deux distances sont importantes D_{min} et D_{max}.

- Lorsque la distance entre les barycentres de 2 clusters est inférieure à D_{min}, ces 2 clusters sont fusionnés en un nouveau. Le barycentre de ce nouveau cluster est ensuite calculé.
- Lorsqu'un la distance entre un nouveau point et le barycentre d'un cluster est inférieure à D_{max}, ce point est ajouté au cluster et le barycentre est recalculé. Dans le cas contraire un nouveau cluster est créé.

Lors de la première génération, chaque point est considéré comme un seul cluster. Une fois les clusters construits, on peut calculer les nouvelles fitness.

La formule suivante peut être utilisée ce qui réduit la complexité obtenue avec le sharing normal en N^2 en une complexité en N log(N), avec N le nombre d'individus.

$$f'(\varepsilon_i) = \frac{f(\varepsilon_i)}{\sum_{j \in C(\varepsilon_i)} s(d(\varepsilon_i, \varepsilon_j))}$$

$C(\varepsilon_i)$ désigne le cluster auquel appartient l'individu ε_i.

Cependant, la formule suivante lui sera préférée, car sa complexité est encore meilleure.

$$f'(\varepsilon_i) = \frac{f(\varepsilon_i)}{n_c \cdot \left(1 - \left(\frac{d(\varepsilon_i, \varepsilon_j)}{2.D_{\max}}\right)^{\alpha}\right)}$$

n_c est le nombre d'individus contenus dans le même cluster que ε_i.

C_i est le centroïde du cluster

α est le coefficient de sensibilité du modèle.

Cet algorithme est donc très puissant cependant, il a fallu introduire 2 nouveaux paramètres dont le choix est délicat et conditionne directement les solutions du problème. Ces paramètres dépendent directement de l'emplacement des individus dans l'espace, or cet emplacement est inconnu au départ.

Pour résoudre cette difficulté, il est possible d'adapter ces deux valeurs en cours de génération. La procédure suivante est employée.

Initialisation $D_{moy} = 0$, $\Delta = 2$ ensuite à chaque génération :

1) Calcul de D_{max} et D_{min}

$$\begin{cases} D_{\max} = \frac{D_{moy}}{\Delta} \\ D_{\min} = \frac{D_{\max}}{3} \end{cases}$$

2) D_{moy} égale distance moyenne des individus par rapport aux barycentres des clusters.

$$D_{moy} = \frac{\sum_{i=1}^{n}\left(\sum_{j=1}^{Nc} d(\varepsilon_i, C_j)\right)}{n.N_c}$$

N_c = Nombre total de clusters

C_j = barycentre du cluster j

3) On calcule N_{opt}, qui est égale au nombre de clusters dont le meilleur individu est supérieur à un certain pourcentage du meilleur individu de la population. Ce pourcentage est appelé *pourcentage de sharing*.

4) Δ est mis à jour, il reste identique à la génération précédente sauf si :

- $N_{opt} / N_c > S_1$ et $\Delta < 100$ alors $\Delta = \Delta * 1.05$
- $N_{opt} / N_c < S_2$ et $\Delta > 1$ alors $\Delta = \Delta * 0.95$

S1 et S2 sont 2 seuils. Les valeurs suivantes seront habituellement prises pour S1 = 0.85 et S2 = 0.75

Si beaucoup de clusters sont suffisamment bons, Δ est augmenté, donc D_{max} et D_{min} sont diminués. Le nombre total de clusters va donc augmenter ce qui va nous permettre de trouver d'autres optimums.

Au contraire, s'ils sont peu nombreux on va diminuer la quantité de clusters en augmentant D_{max} et D_{min}.

Ces techniques de sharing sont très puissantes et très intéressantes car elles permettent d'obtenir plusieurs optima. Cependant elles nécessitent la notion de distance entre individus. Cette notion de distance n'est pas toujours évidente, comme on le verra dans le cas de réseaux de neurones.

2.3.6 Le recuit simulé (Simulated Annealing)

L'introduction de cette technique dans les algorithmes génétiques a été réalisée par Mahfoud et Goldberg [MaGo93]. Après croisement ou mutation, les nouveaux individus seront conservés s'ils sont meilleurs que leurs parents, s'ils sont moins bons une seconde chance leur sera quand même accordée en fonction d'une probabilité p(T).

$$p(T) = \frac{1}{1 + e^{\left(\frac{-|E_i - E_f|}{T}\right)}}$$

E_i fitness du parent et E_f fitness de l'enfant.

Plus la différence entre E_i et E_f est grande plus la probabilité p(T) de garder le parent est grande. En fonction des générations successives, T va diminuer, donc la probabilité de garder les parents va augmenter. Donc au début, si la différence de fitness entre le parent et l'enfant n'est pas trop élevée, la probabilité de garder les enfants est élevée, ce qui favorise l'exploration. En fin de l'algorithme la probabilité de garder les meilleurs individus, c'est à dire les parents, est dominante.

T va donc décroître en fonction du nombre k de générations. La formule suivante est souvent utilisée, avec à l'initialisation T(0) = Ti :

$$T(k+1) = CC * T(k) \text{ avec } 0 < CC < 1.$$

Une amélioration est obtenue en utilisant un recuit géométrique à un palier de basculement ainsi une température de transition Tx est définie ce qui marque une augmentation de la valeur de CC. La diminution de T en fonction des générations est donc plus faible. On aura alors $0 < CC1 < CC2 < 1$

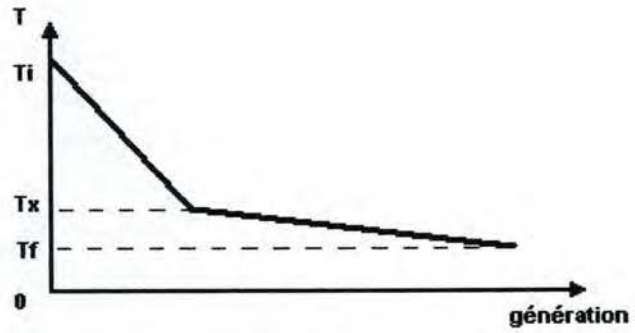


Figure 2-8 : Evolution de la température de recuit en fonction des générations

Les valeurs de T_i, T_x et T_f sont obtenues comme suit

$$T_i = \frac{-\Delta f \max}{\theta_k} \quad avec\ k = 0.75$$

$$T_x = \frac{-\Delta f \max}{\theta_k} \quad avec\ k = 0.99$$

$$T_f = \frac{-\Delta f \min}{\theta_k} \quad avec\ k = 0.99$$

$$\text{avec } \theta_k = \ln\left(\frac{1}{\frac{1}{k} - 1}\right)$$

Avec Δf_{min} et Δf_{max} les écarts minimums et maximums des fitness dans la population initiale.

Pour chacun des paliers le nombre de générations nécessaires à la stabilisation du processus est obtenu par les 2 formules suivantes

$$N1 = \frac{\ln\left(\frac{T_x}{T_i}\right)}{\ln(CC1)} \qquad N2 = \frac{\ln\left(\frac{T_f}{T_x}\right)}{\ln(CC2)}$$

Le nombre optimal de générations est donc N = N1+N2.

2.3.7 Mutation adaptative

Un inconvénient majeur des algorithmes génétiques est leur mauvaise convergence locale. Pour les problèmes à variables réelles, une modification est cependant possible afin d'améliorer la convergence. Le principe de cette modification repose sur une adaptation de l'étape de mutation. L'opérateur de mutation d'origine est en fait l'addition d'un bruit gaussien à l'élément de la population choisi aléatoirement. Le problème est de choisir le bon bruit gaussien à ajouter. Si ce bruit est trop petit, le déplacement dans l'espace empêche de sortir des zones de minimum local. Si ce bruit est trop grand l'algorithme trouvera la bonne zone de l'optimum global mais sera incapable de converger localement vers cet optimum. La modification à apporter est donc d'adapter le bruit gaussien au fur et à mesure des générations. On va donc calculer l'écart moyen pour chacune des coordonnées entre le meilleur élément de la génération *n* et tous les autres éléments. Ce nombre est alors utilisé comme écart type du bruit gaussien sur la coordonnée en question à la génération *n+1*.

3 Chapitre 2 : Réseaux de neurones

3.1 Historique

Les premiers travaux sur les réseaux de neurones remontent à 1943, par la publication des résultats de McCulloch et Pitts [MCPi43]. Ils ont modélisé le fonctionnement du système nerveux à partir d'éléments abstraits, ayant les propriétés des neurones biologiques connues à l'époque. Ce sont eux les premiers à montrer que des réseaux de neurones formels simples peuvent réaliser des fonctions logiques, arithmétiques et symboliques complexes. Ces travaux furent suivis en 1949 par ceux de D. Hebb, qui donna aux réseaux de neurones un premier mécanisme d'apprentissage. La combinaison de ces deux travaux donne ainsi une structure très simple, un neurone et une règle d'apprentissage.

En 1959, F. Rosenblatt développa le modèle du Perceptron. Il construit le premier neuro-ordinateur basé sur ce modèle et l'appliqua au domaine de la reconnaissance de formes. Ce modèle est une première tentative du neurone orienté vers le traitement automatique de l'information.

En 1960, B. Widrow développa le modèle Adaline (Adaptive Linear Element). Dans sa structure, le modèle ressemble au Perceptron, cependant la loi d'apprentissage est différente. Celle-ci est à l'origine de l'algorithme de rétropropagation du gradient, très utilisé aujourd'hui avec les Perceptrons multicouches.

En 1969, M. Minsky et S. Papert publient une analyse rigoureuse des propriétés du Perceptron, qui met en évidence les limitations théoriques du modèle. Ces limitations concernent notamment l'impossibilité théorique de traiter avec ce modèle, des problèmes non linéaires. Ils étendent implicitement ces limitations à tout modèle de réseaux de neurones artificiels. Ils montrent cependant que les Perceptrons multicouches peuvent lever ces problèmes, mais n'apportent pas de réponses aux difficultés posées par l'apprentissage des couches internes de ces réseaux. Cette publication a pour conséquence de diminuer fortement les subsides et les recherches dans le domaine.

Entre 1969 et 1982, toutes les recherches ne sont pas interrompues. De grands noms travaillent toujours durant cette période comme S. Grossberg et T. Kohonen. S. Grossberg publie des travaux sur la théorie de la résonance adaptative [Gro76] qui tente d'inclure des mécanismes d'attention dans des modèles de réseaux à deux couches rebouclées. Kohonon propose un modèle d'auto-organisation qui manifeste des capacités de développement d'une organisation à partir d'une stimulation seule [Koh82].

En 1982, grâce à la publication de J. Hopfield [Hop82], on obtient à nouveau une croissance d'intérêt pour les réseaux de neurones artificiels. Il présente une théorie du fonctionnement et des possibilités des réseaux de neurones. La présentation de son

article est très anticonformiste, car à l'inverse de tous les travaux présentés jusque là, son modèle fixe préalablement le comportement à atteindre et construit, à partir de là, la structure et la loi d'apprentissage correspondant au résultat escompté.

En 1983, la machine de Boltzmann est le premier modèle connu apte à traiter de manière satisfaisante les limitations recensées dans le cas du Perceptron. Mais l'utilisation pratique et en particulier la convergence de l'algorithme, est extrêmement longue.

C'est en 1985 que la méthode de rétropropagation du gradient est découverte et que les limitations du Perceptron mises en avant par M. Minsky sont enfin levées. On doit cette découverte à plusieurs groupes de chercheurs indépendants dont le principal est constitué par Cun Y. L., Rumelhart D., G. Hinton [Cun85].

Aujourd'hui le domaine des réseaux de neurones est toujours en plein développement, plus de 200 compagnies sont impliquées dans des développements d'applications de réseaux de neurones.

3.2 Le réseau de neurones biologiques

Les réseaux de neurones sont inspirés des neurones biologiques. Le neurone biologique est constitué de trois parties :

- Les entrées, *dendrites*
- Le noyau, *corps cellulaire*
- La sortie, *l'axone.*

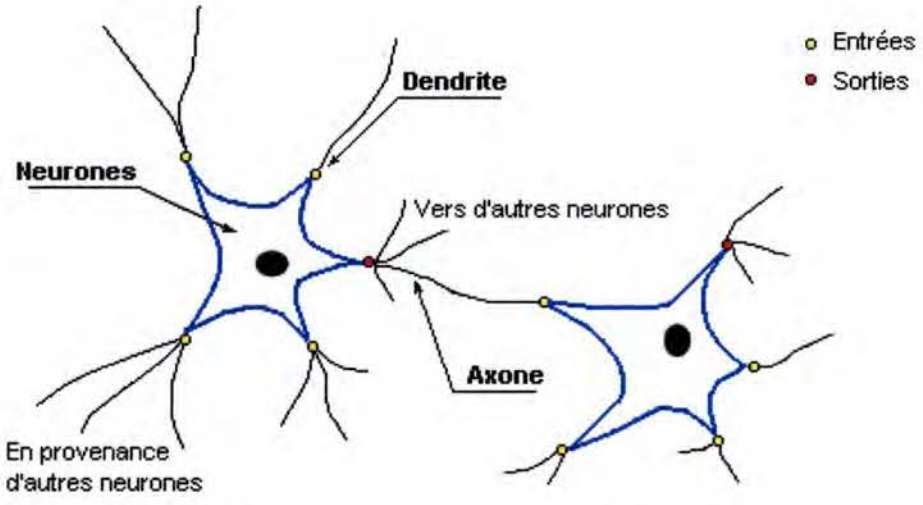


Figure 3-1 : Représentation simplifiée de 2 neurones biologiques

Les neurones reçoivent les signaux (impulsions électriques) par des extensions très ramifiées de leur corps cellulaire, les dendrites et envoient l'information par de longs prolongements, les axones. Les impulsions électriques sont régénérées pendant le

parcours le long de l'axone. La durée de chaque impulsion est de l'ordre d'1 ms et son amplitude d'environ 100 mvolts.

Les contacts entre deux neurones, de l'axone à une dendrite, se font par l'intermédiaire des *synapses*. Lorsqu'un potentiel d'action atteint la terminaison d'un axone, des neuromédiateurs sont libérés et se lient à des récepteurs post-synaptiques présents sur les dendrites. L'effet peut être excitateur ou inhibiteur.

Chaque neurone intègre en permanence jusqu'à un millier de signaux synaptiques. Ces signaux n'opèrent pas de manière linéaire (effet de seuil).

Le cerveau contient environ 100 milliards de neurones. La vitesse de propagation des influx nerveux est de l'ordre de 100 m/s. Cette vitesse est bien inférieure à la vitesse de transmission de l'information dans un circuit électronique. Il existe de quelques centaines à plusieurs dizaines de milliers de contacts synaptiques par neurone. Le nombre total de connexions est estimé à environ 10^{15}.

La connectique du cerveau ne peut pas être codée dans un "document biologique", tel l'ADN, pour de simples raisons combinatoires. La structure du cerveau provient donc en partie des contacts avec l'environnement. L'apprentissage est donc indispensable à son développement. On observe une grande plasticité de l'axone, des dendrites et des contacts synaptiques. Celle-ci est surtout très importante après la naissance, on a observé chez le chat un accroissement des contacts synaptiques de quelques centaines à 12 000 entre le $10^{ème}$ et le $35^{ème}$ jour. Cette plasticité est conservée tout au long de l'existence. Les synapses entre des neurones qui ne sont pas simultanément actifs sont affaiblies puis éliminées.

3.3 Le réseau de neurones artificiels

Un réseau de neurones artificiels est un ensemble de petites unités, qui communiquent entre elles à l'aide de signaux qu'elles s'envoient. Ces signaux sont envoyés par des connexions.

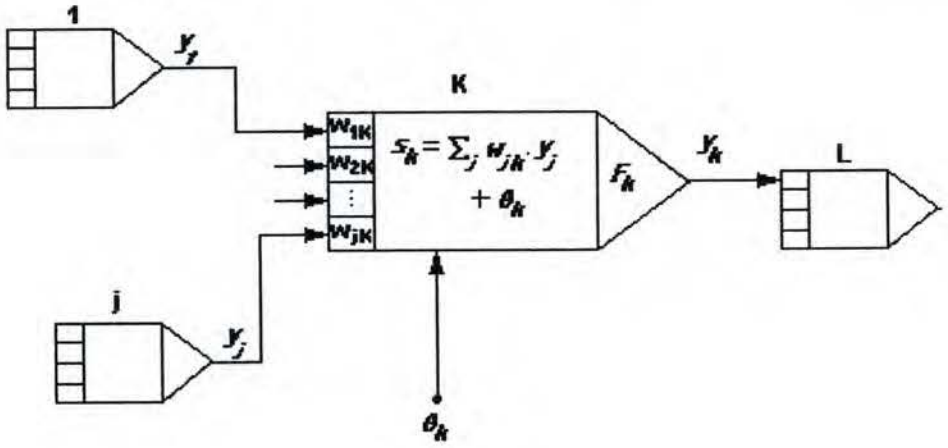


Figure 3-2 : Représentation d'un réseau de neurones artificiels

Les différents éléments modélisant un réseau de neurones classiques sont :

- Les unités du réseau que l'on appelle : neurones ou cellules
- L'état d'activation y_k de chaques neurones qui sont équivalents aux signaux de sortie des unités.
- Les connexions entre les unités possédant chacune un poids w_{jk} qui détermine l'effet qu'a le signal de l'unité j sur l'unité k.
- Les unités possèdent parfois une entrée externe supplémentaire : Le biais ou "offset" θ_k
- Une règle de pondération est affectée aux unités, elle détermine l'entrée effective s_k de l'unité sur base des entrées externes.
- Une fonction d'activation F_k, qui détermine le nouveau niveau d'activation sur base de l'entrée effective s_k.

3.3.1 Les neurones

Chacune de ces unités effectue un travail élémentaire, elle reçoit des signaux d'entrée provenant de ses voisins ou de sources extérieures, qu'elle transforme en signal de sortie qui est propagé vers les autres unités.

Dans ce système il est utile de distinguer 3 types différents d'unité :

- Les unités d'*entrée* qui reçoivent leurs signaux de l'extérieur du réseau de neurones,
- Les unités *cachées* qui ont leurs entrées et leurs sorties inclues dans le réseau de neurones.
- les unités de *sortie* qui envoient des données à l'extérieur du réseau de neurones

3.3.2 Les connexions entre unités

L'entrée effective s_k est très souvent fournie par une combinaison additive des différents signaux d'entrées externes. L'entrée totale est donc la somme pondérée des différents signaux d'entrées externes plus le biais θ_k :

$$s_k(t) = \sum_j w_{jk}(t).y_j(t) + \theta_k(t)$$

Une contribution positive w_{jk} est considérée comme une excitation et une contribution négative w_{jk} est considérée comme une inhibition. Dans certains cas d'autres règles de combinaison plus complexes peuvent être envisagées.

3.3.3 Activation et règle de sortie

Une règle qui va donner l'activation y_k sur base du signal effectif s_k.est nécessaire :

$$y_k(t) = F_k(S_k(t)) = F_k\left(\sum_j w_{jk}(t).y_j(t) + \theta_k(t)\right)$$

Cette règle F_k est souvent une fonction croissante. Les différentes fonctions F_k les plus fréquentes sont : la fonction à seuil, la fonction linéaire par morceau ou la sigmoïde

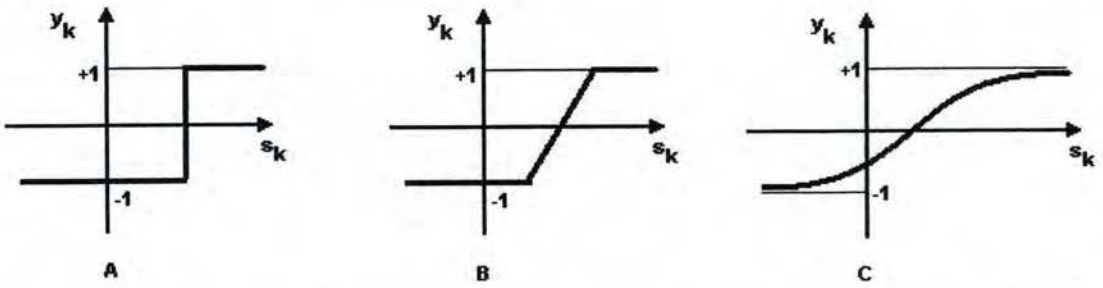


Figure 3-3 : Fonctions à seuil (A), linéaire par morceau (B) et sigmoïde (C)

3.4 Topologie des réseaux

Il y a 2 grands types de réseaux : les réseaux Feed-forward et les réseaux récurrents

- Les réseaux *Feed-forward* : Le flux de données va strictement des neurones d'entrée vers les neurones de sortie sans retour en arrière. Exemples classiques: Le Perceptron, l'Adaline ou le réseau multicouches.

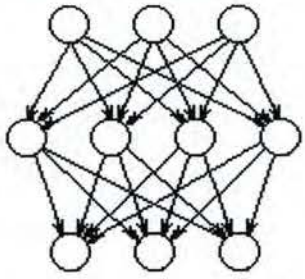


Figure 3-4 : Réseau multicouches

- Les réseaux *Récurrents* : Le réseau contient des connexions de retour, il faut donc attendre plusieurs cycles avant d'avoir une stabilisation des valeurs du réseau. Ce type de réseau est donc dynamique. Des exemples classiques sont les réseaux proposés par Kohonen [Koh77] et Hopfield [Hop82].

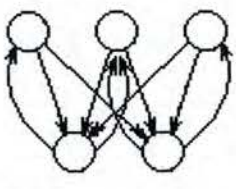


Figure 3-5 : Réseau récurrent

3.5 Apprentissage des réseaux de neurones

Une application typique des réseaux de neurones est la répartition d'individus dans 2 classes différentes. Avant de pouvoir utiliser un réseau, il faut effectuer une phase d'apprentissage. Cette phase a pour but de trouver les poids du réseau qui satisferont l'application désirée. Pour cette phase d'apprentissage, une série de vecteurs d'entrée pour lesquels les sorties réelles sont connues est nécessaire. Les sorties sont calculées avec le réseau pour ces différents vecteurs d'entrée. Les poids du réseau sont adaptés pour que les sorties calculées correspondent au mieux aux sorties réelles. Cette adaptation est basée sur des règles d'apprentissage.

Virtuellement toutes les règles d'apprentissage peuvent être considérées comme des variantes de la règle d'apprentissage de Hebb [Heb49]. L'idée de base est que si deux unités j et k sont actives simultanément, le poids de leurs connexions doit être renforcé. Si k reçoit une entrée de j, une version simplifiée de la règle d'apprentissage de Hebb suggère une modification du poids w_{jk} comme suit.

$$\Delta w_{jk} = \gamma . y_j . y_k$$

avec γ une constante de proportionnalité représentant le taux d'apprentissage.

Une autre règle communément utilisée est la *règle de Widrod-Hoff* ou règle *des deltas*. Elle utilise la différence entre l'activation désirée d_k et l'activation obtenue y_k.

$$\Delta w_{jk} = \gamma . y_j . (d_k - y_k)$$

3.6 Réseaux de neurones à une seule couche

Le Perceptron et l'Adaline sont deux types de réseaux à une seule couche.

3.6.1 Le Perceptron

Le Perceptron est un réseau de neurones avec une fonction d'activation à seuil.

Prenons un exemple avec 2 entrées et un biais.

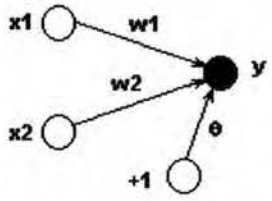


Figure 3-6 : Neurones avec 1 seule couche avec deux entrées et un biais

L'activation de sortie du neurone est fonction des entrées comme suit :

$$y = F\left(\sum_{i=1}^{2} w_i . x_i + \theta \right)$$

La fonction d'activation F choisie pour cette section est une fonction d'activation à seuil définie comme suit :

$$F(s)=\begin{cases} 1 & si\ \ s\geq 0 \\ -1 & si\ \ s<0 \end{cases}$$

La sortie de ce réseau est donc +1 ou -1 en fonction des entrées. Le réseau peut maintenant servir pour la classification. Si le total des entrées est positif, l'individu est assigné dans la classe +1, sinon l'individu est assigné dans la classe -1. La séparation entre ces deux classes est une ligne droite donnée par l'équation :

$$w_1.x_1 + w_2.x_2 + \theta = 0$$

Ou sous une autre forme $$x_2 = -\frac{w_1}{w_2}.x_1 - \frac{\theta}{w_2}$$

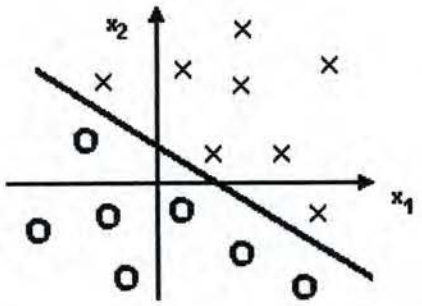


Figure 3-7 : Représentation géométrique de la fonction de discrimination

3.6.2 Règle d'apprentissage du Perceptron

Supposons que nous avons un ensemble d'individus pour l'apprentissage du réseau constitué d'un vecteur d'entrée x et de la sortie désirée d(x). La règle d'apprentissage du Perceptron est très simple :

1. Choisir aléatoirement les poids des connexions
2. Sélectionner un des vecteurs d'entrée de l'ensemble des individus d'apprentissage.
3. Si $y \neq d(x)$ alors le Perceptron donne une réponse incorrecte, donc il faut modifier tous les poids des connexions comme suit :

$$\Delta w_i = d(x).x_i$$

4. Retour à l'étape 2.

Notons que cette procédure est similaire à la règle de Hebb, la seule différence est que si le réseau répond correctement, aucune pondération w_i n'est modifiée.

Remarque: le biais θ (offset) sera également modifié, comme les autres poids w_i, en prenant $x_0 = 1$.

3.6.3 L'Adaline (Adaptive linear element)

L'Adaline est un réseau de neurones avec une fonction d'activation sigmoïde.

Une importante généralisation de la règle d'apprentissage du perceptron a été présentée par Widrow et Hoff, procédure d'apprentissage des moindres carrés, aussi connue comme règle des deltas.

Le problème est de déterminer les coefficients w_j avec j=0,1,...,n de manière telle que les réponses Entrées-Sorties soient correctes, pour un nombre élevé d'individus pris dans l'ensemble des individus d'apprentissage. Si un fitting exact n'est pas possible, la moyenne des erreurs doit être minimisée (au sens des moindres carrés).

Description de la méthode des deltas, avec une activation linéaire.

$$y = \sum_j w_j . x_j + \theta$$

La fonction d'erreur E à minimiser pour les individus est :

$$E = \sum_p E^p = \frac{1}{2} \sum_p \left(d^p - y^p \right)^2$$

Avec E^p erreur pour un des individus p du groupe d'apprentissage, qui est égale à la différence entre la valeur de sortie désirée d^p et la valeur obtenue y^p.

Pour effectuer cette minimisation, la méthode de descente du gradient va être utilisée :

$$\Delta_p w_j = -\gamma \frac{\partial E^p}{\partial w_j}$$

avec γ est la constante de proportionnalité.

La dérivé de la fonction à minimiser est :

$$\frac{\partial E^p}{\partial w_j} = \frac{\partial E^p}{\partial y^p} \frac{\partial y^p}{\partial w_j}$$

Et avec la fonction d'activation linéaire, on a : $\frac{\partial y^p}{\partial w_j} = x_j$

et $$\frac{\partial E^p}{\partial y^p} = -\left(d^p - y^p \right)$$

donc $$\Delta_p w_j = \gamma . \delta^p . x_j \quad avec \quad \delta^p = d^p - y^p$$

3.6.4 Le problème du XOR (ou exclusif)

Le modèle à couche unique a ouvert la voie à beaucoup d'applications, cependant, ce modèle a des limites. Ces limites ont été mises en évidence par M. Minsky and S. Papert [MiPa69]. Les réseaux de neurones sans couche cachée, peuvent représenter

par exemple les fonctions AND, OR, mais pas la fonction XOR. Comme on peut le voir sur la figure 3.8. En effet pour le XOR il est impossible de placer une droite dans le plan pour séparer les individus en 2 classes.

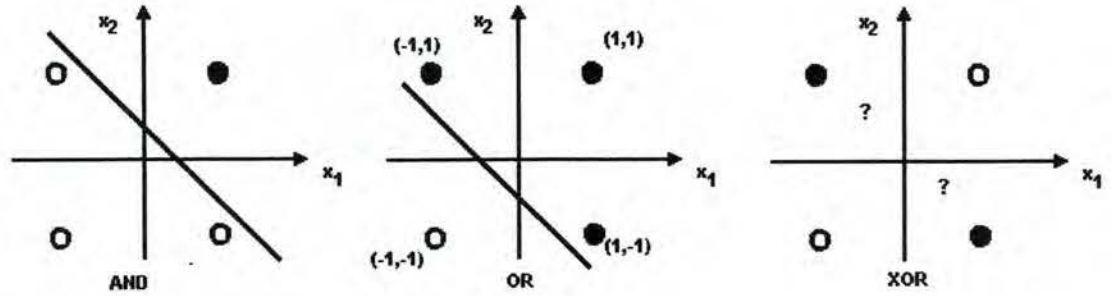


Figure 3-8 : Représentation géométrique du problème XOR

Ce problème est résolu par l'usage de réseaux à une couche cachée. Cependant il apparaît un nouveau problème : comment effectuer l'apprentissage du réseau et en particulier de la couche cachée. Ce problème a été résolu par la généralisation de l'algorithme de *back-propagation* (voir point suivant).

En conclusion, les réseaux sans couche cachée ne peuvent être utilisés que pour faire de la classification linéaire d'individus.

3.7 Réseaux multicouches Feed-forward

Comme nous venons de le voir dans le paragraphe précédent, le réseau à une seule couche a de sérieuses restrictions. Ces restrictions peuvent être levées en utilisant un réseau à plusieurs couches. Le problème étant alors de trouver un algorithme d'apprentissage des couches cachées. La réponse à cette question a été proposée à la fois par Rumelhart, Hinton and Williams en 1986 [Rum86] et un peu plus tôt par Werbos, Parker et Cun en 1985 [Cun85]. Cette méthode est la *règle des deltas généralisée.*

3.7.1 La règle des deltas généralisée (*Backpropagation* BP du gradient)

Avec la fonction d'activation suivante : $y_k^p = F\left(s_k^p\right)$

dans laquelle : $s_k^p = \sum_j w_{jk}.y_j^p + \theta_k$

θ_k = offset du k^{ème} neurone

w_{ij} = poids de la connexion entre les neurones j et k

s_k^p est la valeur de l'activation du k^{ème} neurone

y_k^p est la valeur de sortie du $k^{ème}$ neurone, calculée pour l'individu p du groupe d'apprentissage.

Pour une généralisation correcte de la règle des deltas, nous devons avoir :

$$\Delta_p w_{jk} = -\gamma \frac{\partial E^p}{\partial w_{jk}}$$

E^p est la mesure de l'erreur définie comme étant l'erreur quadratique totale sur les N_0 nœuds de la couche de sortie, pour l'individu p appartenant à l'ensemble des individus du groupe d'apprentissage.

$$E^p = \frac{1}{2}\sum_{0=1}^{N_0}\left(d_0^p - y_0^p\right)^2$$

d_0^p est la sortie désirée du nœud O de la couche de sortie, lorsque l'individu p est sélectionné. La somme totale des erreurs (au sens des moindres carrés) de tous les individus appartenant au groupe d'apprentissage est $E = \sum_p E^p$.

Ensuite il faut trouver les minimums des E^p en ajustent les poids w_{jk}

$$\frac{\partial E^p}{\partial w_{jk}} = \frac{\partial E^p}{\partial s_k^p}\frac{\partial s_k^p}{\partial w_{jk}}$$

$$\text{avec} \quad \frac{\partial s_k^p}{\partial w_{jk}} = y_j^p$$

Posons $\delta_k^p = -\frac{\partial E^p}{\partial s_k^p}$ et on obtient $\Delta_p w_{jk} = \gamma.\delta_k^p.y_j^p$ qui est équivalent à la règle des deltas décrit au chapitre précédent.

Il faut maintenant généraliser cette formule pour l'apprentissage des couches cachées. Pour calculer δ_k^p nous allons appliquer la règle en réécrivant cette dérivée partielle comme le produit de 2 facteurs, le premier facteur reflétant le changement de l'erreur E^p en fonction de la sortie y_j^p et le second facteur reflétant le changement de la sortie y_j^p en fonction de changement de l'entrée s_k.

$$\delta_k^p = -\frac{\partial E^p}{\partial s_k^p} = -\frac{\partial E^p}{\partial y_k^p}\frac{\partial y_k^p}{\partial s_k^p}$$

Calculons le second facteur qui est égal à,

$$\frac{\partial y_k^p}{\partial s_k^p} = F'\left(s_k^p\right)$$

C'est en fait la dérivée de la fonction F pour la $k^{ème}$ unité, évaluée pour l'entrée s_k^p.

Pour calculer le premier facteur il faut considérer 2 cas différents

Le premier cas - l'unité k est une unité de sortie : k = O, dans ce cas on a

$$\frac{\partial E^p}{\partial y_0^p} = -\left(d_0^p - y_0^p\right)$$

Ce qui est identique à la règle des deltas classique.

$$\delta_0^p = \left(d_0^p - y_0^p\right) F'\left(s_0^p\right)$$

Le deuxième cas - l'unité k est dans le réseau : k = h, la mesure de l'erreur peut être écrite comme une fonction de l'entrée du réseau vers la couche de sortie;

$$E^p = E^p\left(s_1^p, s_2^p, \ldots, s_j^p, \ldots\right)$$

ensuite.

$$\frac{\partial E^p}{\partial y_h^p} = \sum_{0=1}^{N_0} \frac{\partial E^p}{\partial s_0^p} \frac{\partial s_0^p}{\partial y_h^p} = \sum_{0=1}^{N_0} \frac{\partial E^p}{\partial s_0^p} \frac{\partial}{\partial y_h^p} \sum_{j=1}^{N_h} w_{k0} . y_j^p = \sum_{0=1}^{N_0} \frac{\partial E^p}{\partial s_0^p} w_{h0} = -\sum_{0=1}^{N_0} \delta_0^p \, w_{h0}$$

Ce qui donne :

$$\delta_h^p = F'\left(s_h^p\right) . \sum_{0=1}^{N_0} \delta_0^p \, w_{h0}$$

En utilisant cette équation récursivement, on peut calculer la correction des deltas pour toutes les unités du réseau au départ de la couche de sortie. Cette procédure constitue la *règle des deltas généralisée* pour les réseaux feed-forward d'unité non linéaire.

3.7.2 Taux d'apprentissage γ et *Momentum* α

La vraie descente du gradient requière des pas infinitésimaux. La constante de proportionnalité est le taux d'apprentissage γ. Pour des raisons pratiques, ce coefficient est pris aussi grand que possible sans toutefois atteindre l'oscillation. Un moyen pour éviter l'oscillation pour des γ élevés, est de changer les poids $\Delta w_{ij}(t+1)$ en fonction du changement de poids précédent $\Delta w_{ij}(t)$, multiplié par un terme, le *momentum* α.

$$\Delta w_{jk}(t+1) = \gamma . \delta_k^p . y_j^p \quad + \alpha \, . \, \Delta w_{jk}(t)$$

α est généralement prit entre 0.5 et 1 avec γ entre 0.25 et 0.5.

La figure 3.9, montre l'effet de la descente du gradient dans l'espace des poids.

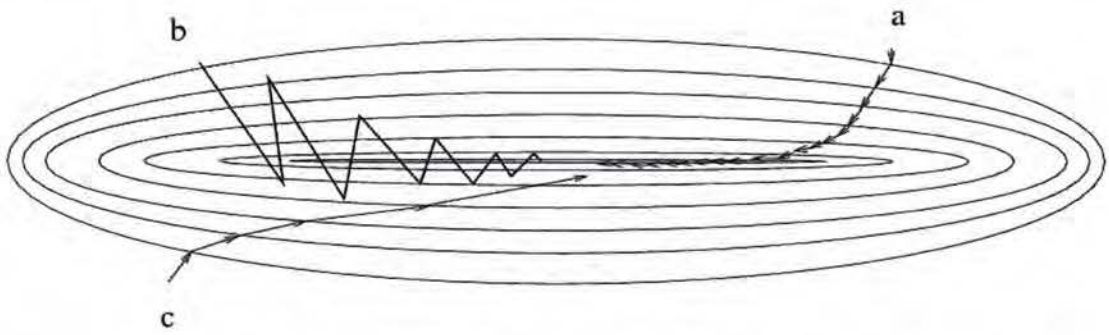


Figure 3-9 : a) Pour un faible taux d'apprentissage γ, b) pour un grand γ (noter les oscillations) c) avec un grand γ et l'addition du terme du momentum α

3.7.3 Apprentissage par succession d'individus uniques

En théorie, l'algorithme de rétropropagation du gradient effectue la descente du gradient sur base de l'erreur totale E, après avoir calculé la somme des erreurs Ep obtenues pour chacun des individus du groupe d'apprentissage. Dans certains cas on effectue la descente du gradient, après chaque calcul d'erreur E^p d'un individu. Il existe des indications empiriques, qui montrent que cette méthode converge plus rapidement. Mais il faut cependant faire attention, car en répétant successivement les séquences des mêmes individus, le réseau peut se focaliser sur les premiers individus. On peut contourner cet inconvénient en utilisant une méthode d'apprentissage incluant une permutation des individus fournis lors de la phase d'apprentissage.

3.7.4 Déficience de la méthode de *back-propagation*

En dépit du succès apparent de l'algorithme d'apprentissage de rétropropagation du gradient, il y a certains aspects qui font que l'algorithme ne garantit pas une convergence universelle. La plupart de ces dysfonctionnements sont dus au long processus d'apprentissage. Cela peut être le résultat d'un mauvais choix du taux d'apprentissage γ ou du *momentum* α. Beaucoup d'algorithmes avancés, basés sur l'apprentissage de rétropropagation du gradient, ont des méthodes optimisées d'adaptation du taux d'apprentissage γ et du *momentum* α. En plus de ce problème, des défaillances d'apprentissage peuvent intervenir, causées par 2 autres problèmes : la *paralysie du réseau* et les *minima locaux*.

La paralysie du réseau

Lors de l'apprentissage les valeurs des poids peuvent prendre des valeurs très élevées (positives ou négatives). Le total des entrées des unités cachées ou des unités de sortie peut donc atteindre des valeurs très élevées (par exemple avec des sigmoïdes). La fonction d'activation F_k donne alors des signaux de sortie très proche de 0 ou très proche de 1. La fonction d'activation F_k pour une sigmoïde est proportionnelle à $y_k^p.\left(1-y_k^p\right)$, qui est donc proche de zéro (pour des y_k proches de 1 ou de 0) et fige par conséquent le processus d'apprentissage.

Les minima locaux

La surface d'erreurs d'un réseau complexe est remplie de montagnes et de vallées. Ce qui fait que l'algorithme du gradient peut converger vers un minimum qui n'est pas l'optimum global. Pour résoudre ce problème, on peut utiliser des méthodes probabilistiques (comme les algorithmes génétiques). Une autre solution est d'augmenter le nombre d'unités cachées. Cette solution aura pour effet d'augmenter la taille de la surface d'erreur et donc de diminuer la probabilité de tomber dans une trappe. Il ne faut cependant pas augmenter ce nombre de manière excessive, sinon on va être confronté à d'autres problèmes (voir point 3.7.7).

3.7.5 Algorithmes avancés

Il y a d'autres algorithmes avancés qui peuvent être utilisés. On notera la méthode des gradients conjugués, utilisant la notion de matrice Hessienne (dérivés secondes). Cette méthode permet d'orienter la direction de recherche vers la plus grande pente, ce qui permet par exemple, de trouver le minimum d'une fonction quadratique à N degrés de liberté, en N pas successifs. Cependant, à l'heure actuelle, aucune de ces méthodes n'a fait de percée aussi significative que la méthode de rétropropagation du gradient.

3.7.6 Les effets du nombre d'individus de l'ensemble d'apprentissage

Supposant un problème d'optimisation simple avec une fonction y=f(x) qui doit être approximée par un réseau de neurone avec une entrée x, 5 neurones cachés, une fonction d'activation sigmoïde et une unité de sortie linéaire. La figure 3.10 montre que, au plus le nombre d'individus pour l'apprentissage est élevé, au mieux la fonction est approximée par le réseau de neurones.

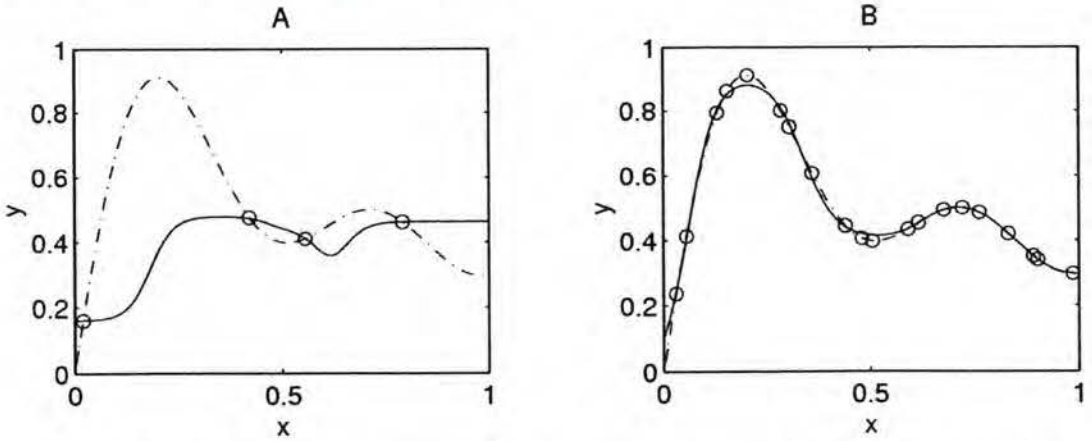


Figure 3-10 : En pointillé la fonction désirée, en trait plein l'approximation par le réseau de neurones. Nombre d'individus pour l'apprentissage (a) 4 et (b) 20

La figure 3.11 montre que le taux d'erreurs sur l'apprentissage augmente lorsque la taille du groupe d'individus d'apprentissage augmente et que ce taux d'erreurs après la phase d'apprentissage, lui, diminue. Si ces 2 courbes ne convergent pas vers la même

valeur, c'est que l'algorithme n'a pas convergé vers le minimum global. En conclusion, une bonne approximation d'une fonction par un réseau de neurone est obtenue par un apprentissage avec un nombre d'individus élevé.

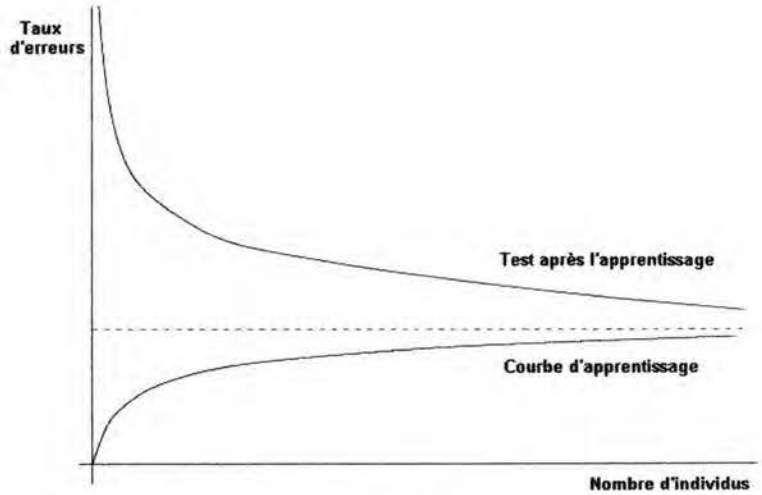


Figure 3-11: Effet de la taille du groupe d'apprentissage sur le taux d'erreurs.

3.7.7 Les effets du nombre de couches cachées

Supposons le même problème d'optimisation qu'au point précédent, mais cette fois-ci on fait varier le nombre d'unités cachées avec un même nombre d'individus pour la phase d'apprentissage (12 individus). La figure suivante permet de comparer l'approximation avec 5 unités cachées (à gauche) et 20 unités cachées (à droite).

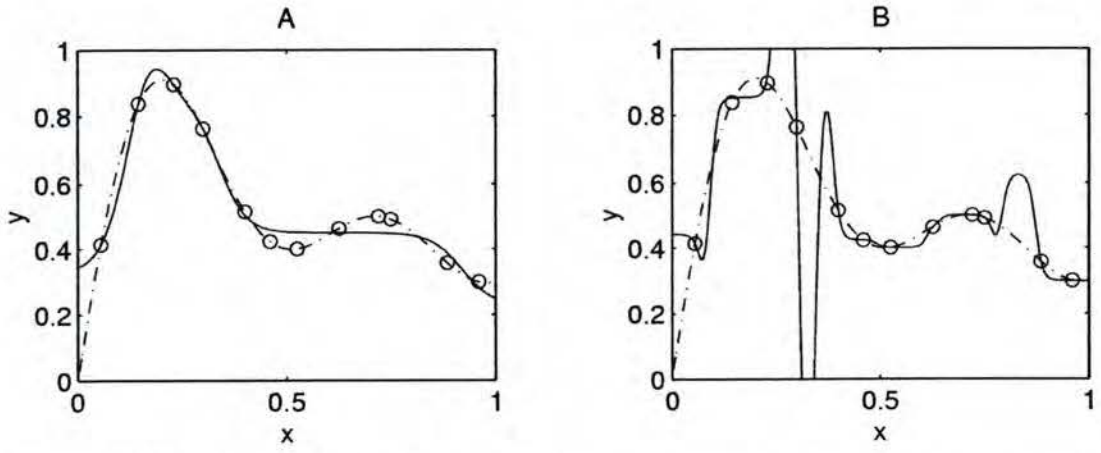


Figure 3-12 : En pointillé la fonction désirée, en traits pleins l'approximation par le réseau. a) 5 unités cachées b) 20 unités cachées

On constate qu'un nombre trop élevé d'unités cachées donne un réseau avec de très mauvaises performances. Ce nombre d'unités cachées ne doit ni être trop petit ni trop grand, figure 3.13.

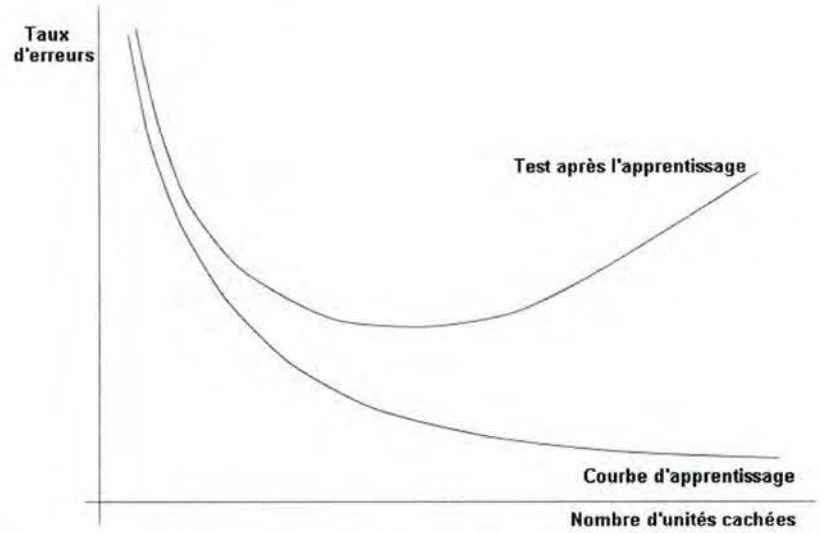


Figure 3-13 : L'erreur moyenne d'apprentissage et l'erreur moyenne sur les individus après apprentissage en fonction du nombre d'unités cachées.

3.8 Les réseaux récurrents

3.8.1 Généralité

Les réseaux récurrents sont des réseaux dans lesquels le flux d'information peut circuler de nœuds en nœuds et revenir au nœud de départ. Ce type de réseaux est donc dynamique et évolue dans le temps jusqu'à ce que ses valeurs se stabilisent.

3.8.2 Le réseau de Hopfield

Ce modèle a été proposé au début des années 1980, il a la particularité d'être dynamique, contrairement à l'Adaline et au Perceptron, dont les sorties sont calculées en une seule passe, après présentation des entrées.

A. Structure

Chaque nœud est connecté à tous les autres nœuds sauf à lui-même ($w_{ii} = 0$), les poids des connexions sont symétriques ($w_{ij} = w_{ji}$). La sortie d'un nœud est égale à 1 si la somme des activations est supérieure ou égale à 0, elle est égale à 0 lorsque cette somme est négative. Voici par exemple un réseau à 3 nœuds.

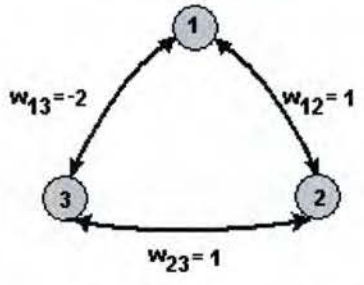


Figure 3-14 : Réseau de Hopfield à 3 nœuds

Après une phase d'initialisation du réseau, un nœud est choisi au hasard et sa sortie est mise à jour. Cette étape de tirage au sort est répétée un certain nombre de fois pour faire évoluer le réseau vers un état stable. La distance de Hamming entre 2 vecteurs de sortie successifs (X1, X2, X3), est à chaque fois au maximum de 1. Pour chaque état possible, on peut calculer quel sera l'état suivant en fonction du nœud qui va être mis à jour.

Etat				Nouveaux Etats après mise à jour du :		
N°	X1	X2	X3	Nœud 1	Nœud 2	Nœud 3
0	0	0	0	4	2	1
1	0	0	1	1	3	1
2	0	1	0	6	2	3
3	0	1	1	3	3	3
4	1	0	0	4	6	4
5	1	0	1	1	7	4
6	1	1	0	6	6	6
7	1	1	1	3	7	6

Ce tableau peut être représenté par un graphe orienté où on indique la probabilité de réalisation d'une transition.

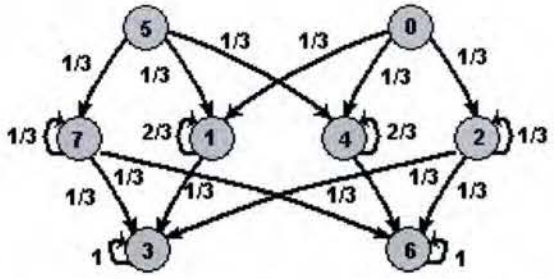


Figure 3-15 Graphe de transition d'états pour une mise à jour Asynchrone

Les états sont représentés de manière telle que l'on tend à se déplacer vers le bas du graphe. On converge soit vers l'état 3, soit vers l'état 6 qui sont les états stables du réseau. Ces états sont appelés les mémoires du réseau, et correspondent respectivement au vecteur de sortie (0,1,1) et (1,1,0).

B. Energie du réseau

La dynamique du réseau est décrite parfaitement à l'aide de la table ou du graphe de transition d'état. Nous allons maintenant représenter ces transitions pas une fonction d'énergie et montrer que le réseau converge toujours vers un état stable d'énergie minimum.

L'idée proposée par Hopfield est d'utiliser la fonction : $e_{ij} = - w_{ij} . x_i . x_j$ pour chaque paire de nœuds. La somme de cette fonction pour l'ensemble du réseau représente l'énergie totale du réseau.

$$E = \sum_{paires} e_{ij} = - \sum_{paires} w_{ij} . x_i . x_j$$

Les états de sortie de l'exemple de la figure 3.15 ont tous la valeur 0, sauf les sorties des états 3 et le 6 qui valent -1 et la sortie de l'état 5 qui vaut à 2.

Sur base des hypothèses $w_{ij} = w_{ji}$ et $w_{ii} = 0$, cette équation peut être réécrite comme suit

$$E = -\frac{1}{2}\sum_{j=1}^{N}\sum_{i=1}^{N} w_{ij}.x_i.x_j$$

Regardons la valeur de l'énergie lors de la mise à jour du nœud k.

$$E = -\frac{1}{2}\sum_{\substack{i\neq k\\ j\neq k}} w_{ij}.x_i.x_j - \frac{1}{2}\sum_{i} w_{ki}.x_k.x_i - \frac{1}{2}\sum_{j} w_{ik}.x_i.x_k$$

comme $w_{ik} = w_{ki}$

$$E = -\frac{1}{2}\sum_{\substack{i\neq k\\ j\neq k}} w_{ij}.x_i.x_j - \sum_{i} w_{ki}.x_k.x_i$$

et pour simplifier la notation remplaçons par S le premier terme du membre de droite

$$E = S - x_k \sum_{i} w_{ki}.x_i$$

la somme ci-dessus est l'activation du nœud K, a^k.

$$E = S - x_k.a^k$$

soit E' la valeur de l'énergie après mise à jour du nœud K et x' la nouvelle sortie.

$$E' = S - x'_k.a^k$$

La variation d'énergie est égale à

$\Delta E = -\Delta x_k.a^k$ avec $\Delta E = E'-E$ et $\Delta x_k = x'_k - x_k$

Il y a deux cas à considérer :

1) $a^k \geq 0$: la sortie passe de 0 à 1 ou reste à 1 donc $\Delta x_k \geq 0$ donc $\Delta E \leq 0$

2) $a^k < 0$: la sortie passe de 1 à 0 ou reste à 0 donc $\Delta x_k < 0$ donc $\Delta E \leq 0$

Dans les 2 cas, à chaque transition la valeur de l'énergie totale du réseau reste constante ou diminue. Comme la valeur de l'énergie est bornée inférieurement par définition étant donné que les poids w_{ij} sont eux-mêmes bornés en valeur absolue, on a alors la preuve que le réseau converge vers un état stable correspondant à un minimum de E.

C. Mise à jour synchrone et asynchrone

Dans la pratique, on utilise 2 grands types de mise à jour différents, la méthode synchrone et la méthode asynchrone. La méthode *synchrone*, pour laquelle les valeurs de tous les neurones sont modifiées au même instant, en d'autres termes, on calcule les nouvelles valeurs dans un ordre sans importance, mais on utilise aucune de celles-

ci avant d'avoir effectué les calculs sur la totalité du réseau. La méthode *asynchrone* (cf ci-dessus) un seul neurone voit la sortie de son état changer à chaque itération: La nouvelle valeur ainsi calculée est utilisée pour la mise à jour suivante d'un autre neurone. L'asynchronisme des mises à jour des neurones est une condition essentielle de la convergence du réseau. Si on reprend l'exemple ci-dessus pour une mise à jour *synchrone*, seuls certains états convergences vers des états stables (comme l'état 1, 3, 4 et 6) les autres convergent vers une configuration instable.

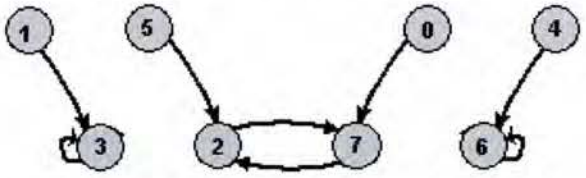


Figure 3-16 : Graphe de transition d'états pour une mise à jour Synchrone

D. Apprentissage

Différentes possibilités sont envisageables pour calculer la matrice des w_{ij} afin de fixer les états stables. La condition de stabilité d'un vecteur y est, pour tout i :

$$F\left(\sum_j w_{jk}.y_j\right) = y_i$$

Si on veut enregistrer un seul vecteur *y(1)*, on peut voir facilement que la condition ci-dessus est réalisée si on prend chaque w_{ij} proportionnel au produit de $y_i(1)$ par $y_j(1)$, *y(1)* est donc stable mais *-y(1)* est stable également. En prenant un vecteur au hasard, il va converger soit vers *y(1)* soit vers *-y(1)*. La notion de bassin d'attraction va donc être introduite. Le bassin d'attraction du vecteur *y(1)* est l'ensemble des vecteurs initiaux qui conduisent au vecteur *y(1)* après convergence.

La principale utilité du modèle de Hopfield réside dans son utilisation en tant que mémoire auto-associative: Une version corrompue ou incomplète du vecteur y(1) est présentée au réseau et on souhaite que la dynamique du réseau converge vers le vecteur *y(1)*. Le but est donc d'avoir des bassins d'attractions les plus larges possibles.

Un cas plus intéressant est l'enregistrement de p vecteurs. Une extension de la règle d'apprentissage de ci-dessus est :

$$w_{ij} = \frac{1}{N}\sum_{k=1}^{P} y_i(k).y_j(k)$$

L'analogie de cette formule avec la règle de Hebb est frappante. Cependant beaucoup de résultats théoriques montrent que cette formule a des limitations sérieuses. Ces limitations sont tant du point de vue du nombre d'états mémorisables, que de la taille des bassins d'attraction. Pour remédier à ces limitations on peut utiliser les autres méthodes classiques d'apprentissage vues pour le neurone isolé (comme par exemple la descente du gradient), car chaque neurone du réseau d'Hopfield peut être considéré comme un neurone isolé à N entrées et à une seule sortie.

3.8.3 Le réseau de Kohonen

A. Introduction

Ce modèle a été inspiré par les résultats de l'étude de Hubel et Witsel en 1947, ils ont montré qu'à deux zones proches dans le cortex visuel correspondent deux zones également proches dans la rétine.

Plusieurs expériences mettent en évidence le fait que cette organisation n'est pas génétique, mais qu'elle se met en place au cours d'une phase d'apprentissage. Suite à ces constatations Kohonen proposa un modèle de carte topologique auto-adaptative. Seules les entrées modifient le processus, l'apprentissage n'est donc plus supervisé mais non-supervisé.

B. Architecture

Voici un exemple de réseau avec un espace d'entrée à deux dimensions et 5 unités.

Le vecteur présenté en entrée est $x = [x_1, x_2, \ldots, x_N]$,

La matrice Wij est composée des vecteurs de poids $W_i = [W_{i1}, W_{i2}, \ldots, W_{iK}]^T$ associés aux différentes entrées i :

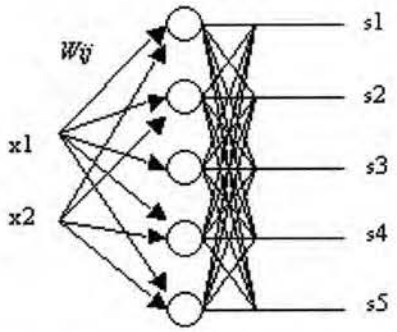


Figure 3-17: Structure d'un réseau de Kohonen à 2 entrées et 5 unités

La première couche d'interconnexions a pour fonction de déterminer l'activité de chaque neurone de sortie, en fonction d'un vecteur de stimuli présenté en entrée. Si K est le nombre de neurones et N le nombre d'entrées du réseau, x un vecteur de stimuli de composant $x = [x_1, x_2, \ldots, x_N]^T$, W la matrice (K, N) des poids du réseau et S le vecteur d'activité des neurones $[s_1, s_2, \ldots, s_K]^T$, alors l'activité des neurones se calcule par : $s(t) = W.x(t)$

La deuxième couche d'interconnexions implante un réseau à inhibitions latérales. Chaque neurone est en relation avec ses voisins selon une fonction d'interaction telle que les poids associés aux connexions entre des neurones physiquement voisins sont élevés. La valeur du lien entre neurones diminue avec la distance à laquelle ils se trouvent. Chaque neurone de la carte de Kohonen est relié à tous les neurones de la carte. Les valeurs des poids sont donc déterminées en fonction de la distance selon une fonction "chapeau mexicain" ou DOG (Difference of Gaussians).

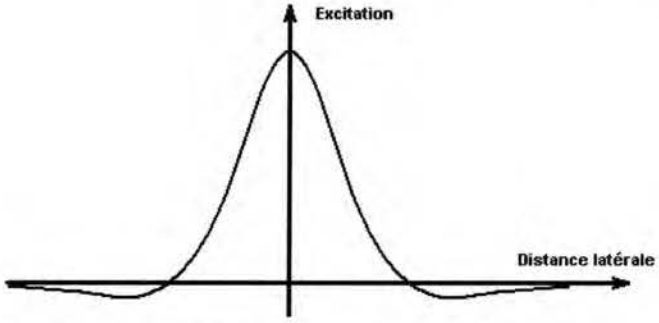


Figure 3-18 : Fonction DOG de l'excitation en fonction de la distance

C. Phase d'apprentissage

1. Les poids W_{ij} sont initialisés aléatoirement avec des faibles valeurs.
2. Un nouveau vecteur x est présenté au réseau.
3. Le nœud *C* dont la distance d_j entre le vecteur présenté en entrée $X = [x_1, x_2, \ldots, x_N]$ et le vecteur de poids $W_C = [W_{C1}, W_{C2}, \ldots, W_{CK}]^T$ est la plus petite, est sélectionné : $$d_j = \|x - W_C\|^2 = \sum_{i=1}^{N} (x_i - W_{ci})^2$$
4. Les vecteurs des poids de ce nœud C et ceux des nœuds se situant dans son voisinage sont modifiés avec la règle d'apprentissage suivante :

$$W_{kj}(t+1) = W_{kj}(t) + \eta(t).\left(x_j(t) - W_{kj}(t)\right)$$

Le coefficient d'apprentissage η est compris entre 0 et 1, sa valeur décroît en fonction du temps et de la distance par rapport au nœud *C* (fonction DOG).

5. Retour à l'étape 2 jusqu'à stabilisation des poids.

D. Phase d'utilisation

On présente un motif particulier au réseau et c'est le neurone ayant la d_i la plus faible qui réagit. La sortie est activée en conséquence.

E. Exemple

Voici un exemple d'un réseau à 2 entrées et une carte de sortie de 8 x 8, à gauche le réseau de départ et à droite la carte complètement formée.

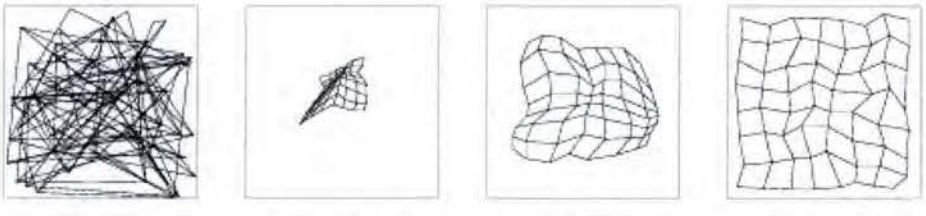


Figure 3.20 : Evolution de la topologie du réseau en fonction de l'apprentissage

4 Chapitre 3 : Applications des algorithmes génétiques aux réseaux de neurones

4.1 Introduction

Les algorithmes génétiques et les réseaux de neurones sont tous les deux inspirés de systèmes biologiques, il n'est donc pas surprenant que les chercheurs dans ces deux domaines, ont tenté de tirer parti de leurs mises en commun.

Les algorithmes génétiques ont été utilisés conjointement aux réseaux de neurones dans 5 grandes catégories d'applications différentes.

Premièrement, les algorithmes génétiques ont été utilisés pour l'**apprentissage des poids** d'un réseau, pour une architecture fixée. Il y a deux grandes catégories d'approches différentes. La première approche est l'utilisation des algorithmes génétiques GA, à la place de la rétropropagation du gradient BP, pour minimiser la fonction d'erreur. La seconde approche est l'utilisation combinée d'algorithmes GA et de méthodes de recherche locale très efficaces, du type BP. Cette dernière approche est appelée apprentissage hybride.

Deuxièmement, les algorithmes génétiques ont également été utilisés pour **trouver l'architecture optimale d'un réseau**, incluant la recherche du nombre de neurones ainsi que les connexions optimales entre ces neurones. Quatre méthodes vont être développées: la méthode d'encodage directe, la méthode d'encodage paramétrique, la méthode d'encodage de grammaire et la méthode par programmation génétique.

Une troisième application est l'utilisation d'algorithmes génétiques pour **trouver une bonne règle d'apprentissage** ainsi que les paramètres associés.

Quatrièmement, les algorithmes génétiques ont été utilisés pour **diminuer le nombre d'entrées** de réseau de neurones, afin de garder uniquement les entrées pertinentes.

Cinquièmement, les algorithmes génétiques ont été utilisés pour **sélectionner les meilleurs éléments des groupes d'apprentissage** lors d'une phase d'apprentissage de réseau de neurones.

Ces cinq applications différentes peuvent être utilisées séparément ou simultanément, la figure 4.1 schématise ces interactions.

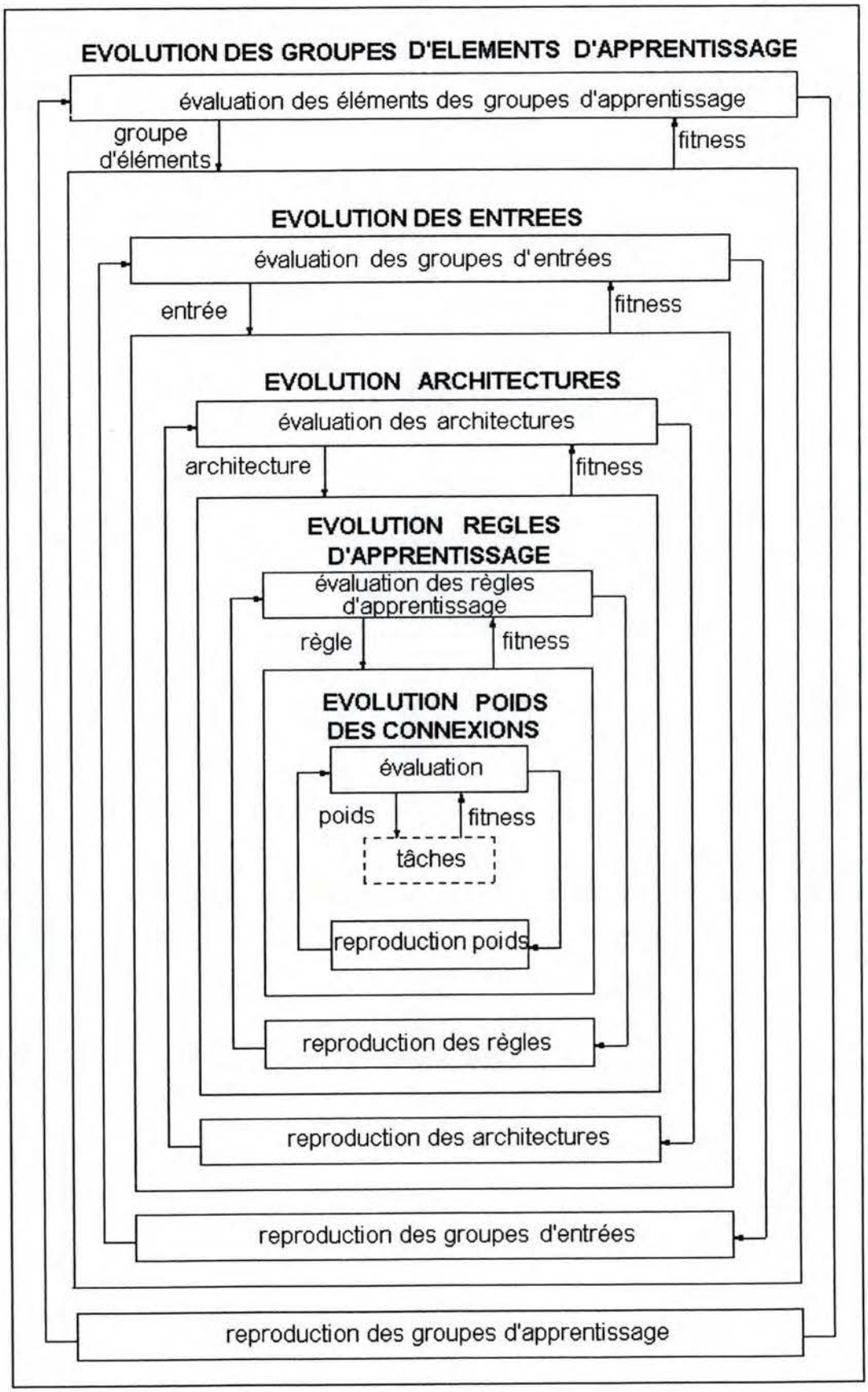


Figure 4-1: Interactions entre les différentes applications des algorithmes génétiques GA aux réseaux de neurones artificiels ANN

4.2 Apprentissage des poids d'un réseau de neurones par algorithme génétique GA

4.2.1 Introduction

L'apprentissage des poids d'un réseau de neurones est généralement obtenu en minimisant une fonction d'erreur. Cette fonction d'erreur est calculée à l'aide d'un groupe d'éléments d'apprentissage pour lesquels les entrées ainsi que les sorties espérées sont connues. La plupart des méthodes d'apprentissage, telle que la rétropropagation du gradient BP, ainsi que les algorithmes de gradients conjugués, sont basés sur la descente le long de la fonction à minimiser. Ces méthodes ont obtenu de grands succès dans de nombreuses applications. Cependant, un des problèmes fréquemment rencontré lors de l'apprentissage de réseaux de neurones complexes (c-à-d lorsque la fonction à minimiser est multimodale), est la convergence vers des sous-optimums locaux. A la fin du processus d'itérations, on n'obtient donc pas l'optimum global de la fonction d'erreur, mais un de ses optima locaux.

Une solution proposée par les chercheurs pour résoudre ce problème est l'utilisation des algorithmes génétiques GA pour effectuer cet apprentissage. Beaucoup d'approches différentes ont été proposées, on peut les classer en deux grandes catégories en fonction de la manière dont l'encodage est réalisé. Cet encodage peut être fait, soit avec des chaînes binaires, soit avec des vecteurs de réels. Les différences principales entre ces deux catégories résident dans les différents opérateurs d'évolution utilisés.

Afin d'améliorer encore cette convergence, les chercheurs ont proposé une approche un peu plus élaborée, l'apprentissage hybride. Cette approche consiste à combiner les GA et la BP, pour bénéficier des avantages combinés de chacune de ces deux méthodes.

Il est important de noter que dans le chapitre qui va suivre, le processus d'optimisation ne porte que sur les poids du réseau de neurones. L'architecture du réseau est fixée au départ, le nombre de neurones ainsi que leurs connexions restent donc inchangés au cours de l'apprentissage. L'optimisation de l'architecture sera étudiée au chapitre suivant.

4.2.2 L'encodage

4.2.2.1 L'encodage binaire

Les premières applications de l'apprentissage des poids de réseaux de neurones par algorithmes génétiques ont été effectuées en encodant les différents poids du réseau à l'aide de chaînes de bits. Ces chaînes de bits sont ensuite concaténées les unes à la suite des autres pour former les individus. L'ordre de concaténation est important comme on le verra au point 4.2.3. Le réseau est donc représenté par cette suite de bits.

L'avantage de cette représentation est sa simplicité, en effet, les opérateurs d'évolutions classiques sont très facilement utilisables.

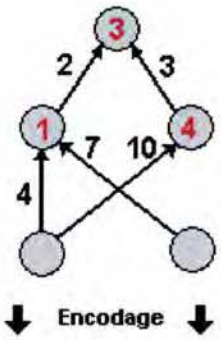


Figure 4-2: Représentation binaire des poids d'un réseau de neurones sur base d'un codage à 4 bits (en rouge les poids des biais)

Lorsqu'on effectue l'encodage de réels à l'aide de chaînes de bits, un problème de taille survient. Ce problème est lié au fait qu'en utilisant un nombre raisonnable de bits pour l'encodage, travailler avec une grande précision et sur une grande plage de réels est incompatible. Si on veut quand même avoir les deux, l'utilisation de longues chaînes de bits est inévitable ce qui rend la convergence lente et parfois même inefficace.

4.2.2.2 L'encodage de réels

Ce type d'encodage transforme le réseau de neurones en un vecteur de réels. Beaucoup d'applications utilisent ce type d'encodage. Les opérateurs d'évolution sont utilisés sur ces vecteurs.

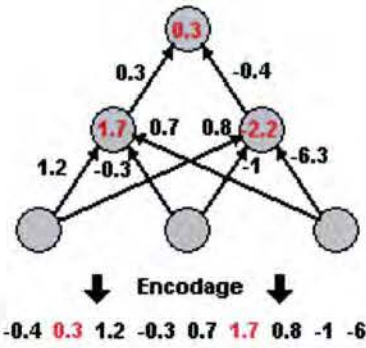


Figure 4-3: Encodage des poids d'un réseau de neurones (en rouge les poids des biais)

4.2.3 Ordre de concaténation

Pour choisir l'ordre de concaténation des poids d'un réseau de neurones, l'heuristique suivante est fréquemment utilisée. Les poids des différentes connexions entrant dans un même nœud sont regroupés, ces groupes sont ensuite mis les uns à la suite des autres. Cette heuristique est utilisable aussi bien avec l'encodage en chaînes de bits,

qu'avec l'encodage en vecteurs de réels. Les bons résultats obtenus avec cette heuristique peuvent se justifier par le fait que les signaux de sortie des neurones sont construits sur base de l'ensemble des signaux d'entrée. Ainsi, pour que l'opérateur de croisement soit bénéfique, il faut éviter de détruire ces arrangements. En effet, si on ne procède pas de la sorte, les valeurs des poids sont réparties aléatoirement dans la chaîne de bits, et la probabilité de séparer ces différentes valeurs lors de croisements est très élevée, ce qui est très pénalisant.

4.2.4 Le problème de la permutation *(Competing Conventions Problem)*

Un problème également connu, lors de l'apprentissage de réseau de neurones par algorithme génétique GA est le problème de la permutation (*Competing Conventions Problem*). Ce problème est lié au fait qu'un même réseau de neurones peut être encodé de plusieurs manières différentes. Si on prend l'exemple de la figure 4.4 et qu'on réarrange la position des nœuds cachés [A B C D], les résultats calculés par ces différents réseaux sont identiques.

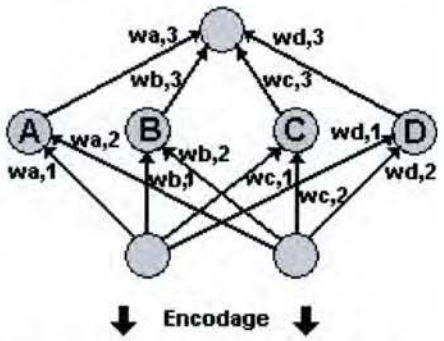


Figure 4-4 : Un simple réseau de neurones.

Dans cet exemple il est possible de construire 24 vecteurs équivalents (*=4!*). Ces vecteurs représentent exactement le même réseau. Lors des opérations d'évolutions, si on utilise le *crossover* à un point, sur 2 de ces 24 vecteurs,

Parent 1 : **[A ⋮ B C D]**

Parent 2 : **[D ⋮ A B C]**

on obtient, par exemple, les deux vecteurs suivants :

Fils 1 : **[A ⋮ A B C]**

Fils 2 : **[D ⋮ B C D]**

On peut constater une perte d'information en effectuant cette opération de croisement sur ces 2 vecteurs permutés. Dans cet exemple, les informations perdues pour le Fils 1 sont les poids $w_{d,1}$ $w_{d,2}$ et $w_{d,3}$ liés au nœud D. Ce problème de la permutation rend donc l'opérateur de croisement très inefficace et ce d'autant plus que le nombre de nœuds par couche cachée est élevé.

Des réponses à ce problème ont été exposées dans la littérature. Une de ces méthodes utilisant les algorithmes génétiques [MoDa89] sera développée au point 4.2.6.2.3.3. D'autres méthodes plus sophistiquées utilisant les algorithmes génétiques combinés à la programmation logique par contraintes (CLP) ont également été proposées [KMMR96] et [KMMR97], mais elles ne seront pas développées ici car elles sortent du cadre de ce travail.

4.2.5 La fonction d'évaluation

Comme nous l'avons vu précédemment, l'utilisation d'algorithme génétique GA nécessite une fonction d'évaluation. Cette fonction d'évaluation permet d'associer à chacun des individus de la population une fitness. L'apprentissage qui est effectué par algorithme génétique GA est un apprentissage supervisé. Pour pouvoir effectuer cet apprentissage supervisé, nous devons disposer d'un groupe d'apprentissage. Ce groupe d'apprentissage est composé d'éléments pour lesquels les différentes entrées ainsi que les sorties espérées sont connues.

Pour chaque individu de la population (c'est-à-dire pour chaque vecteur de poids), on calcule la somme des différences (au sens des moindres carrés) entre les sorties calculées par le réseau de neurones et les sorties espérées de chacun des éléments du groupe d'apprentissage. Cette somme est retournée par la fonction d'évaluation, c'est la fonction d'erreur. Le but de l'algorithme génétique est de minimiser cette fonction.

4.2.6 Apprentissage des poids par algorithme génétique GA

4.2.6.1 Apprentissage avec encodage binaire

Lorsqu'on utilise l'encodage binaire, étant donné sa grande simplicité, les opérateurs d'évolution classiques sont très facilement utilisables. Pour le *crossover*, le croisement à un ou plusieurs points est utilisé. Pour la mutation, la modification aléatoire de bits est utilisée.

4.2.6.2 Apprentissage avec encodage de réels

Lorsqu'on effectue l'apprentissage de réseau de neurones à l'aide d'algorithmes génétiques et que l'on choisit d'effectuer l'encodage avec des vecteurs de réels, les opérateurs d'évolution classiques ne peuvent être utilisés. Les chercheurs comme D. Montana et L. Davis ont donc développé des opérateurs d'évolution particuliers (DMoLDa89). L'application pour laquelle ces deux chercheurs ont été amenés à développer ces opérateurs d'évolution particuliers, est la résolution d'un problème complexe de classification. Cette classification avait pour but le classement de sons sous-marins (des *lofargrammes* similaires aux spectrogrammes), en deux grandes catégories, les sons "intéressants" et les sons "inintéressants".

Le réseau de neurones qui a été choisi pour effectuer cette tâche est le réseau à couches entièrement connectées, composé de quatre entrées, deux couches cachées et d'une sortie. Les deux couches cachées sont constituées de 7 neurones pour la première et de 10 neurones pour la seconde. Ce réseau possède donc 126 poids (en incluant les biais).

Pour effectuer la phase d'apprentissage de ce réseau, ces chercheurs ont utilisé un groupe d'apprentissage de 236 éléments. La population d'individus utilisée par l'GA est constituée de 50 vecteurs de poids.

4.2.6.2.1 La procédure d'initialisation

Les poids des individus de la population initiale sont choisis sur base d'une probabilité de distribution du type $e^{-\|x\|}$. Cette initialisation permet d'obtenir des valeurs centrées autour de 0. Cette probabilité de distribution reflète l'observation empirique de ces chercheurs qui ont mis en évidence, le fait que la solution optimale tend à contenir des poids avec de faibles valeurs absolues.

4.2.6.2.2 Le premier opérateur d'évolution : La mutation

4.2.6.2.2.1 Mutation non biaisé des poids (Unbiased-Mutate-Weights)

Sur base d'une probabilité p=0.1, les éléments des vecteurs de poids sont remplacés par des valeurs aléatoires choisies à l'aide de la probabilité de distribution du type $e^{-\|x\|}$ (distribution centrée autour de 0).

4.2.6.2.2.2 Mutation biaisé des poids (Biased-Mutate-Weights)

Sur base d'une probabilité p=0.1, est ajoutée aux éléments des vecteurs de poids, une valeur aléatoire choisie à l'aide de la probabilité de distribution du type $e^{-\|x\|}$ (distribution centrée autour de l'ancienne valeur du poids). Ce type de mutation est susceptible d'être meilleure que la précédente, car une nouvelle valeur centrée autour de l'ancienne a plus de chance d'être meilleure qu'une nouvelle valeur centrée autour de 0.

4.2.6.2.2.3 Mutation des nœuds ("Mutate-Nodes")

Cet opérateur sélectionne aléatoirement *n* nœuds, différents des nœuds d'entrées. Pour chacune des entrées de ces nœuds sélectionnés, une valeur aléatoire choisie à l'aide de la distribution de probabilité du type $e^{-\|x\|}$ leur est ajoutée (distribution centrée autour de l'ancienne valeur du poids).

4.2.6.2.2.4 Comparaison des différents opérateurs de mutation

En comparant la mutation non-biaisé des poids, la mutation biaisé des poids et la mutation des noeuds, voir figure 4.5, on obtient par ordre décroissant de performance:

1) La mutation des nœuds
2) La mutation biaisé des poids
3) La mutation non-biaisé des poids.

Les résultats obtenus sont en accord avec ce que l'on s'attendait à obtenir.

Cette comparaison a été effectuée en utilisant dans les trois cas, les mêmes paramètres excepté le type de mutation. Pour l'opérateur d'évolution de croisement, le croisement des poids a été utilisé.

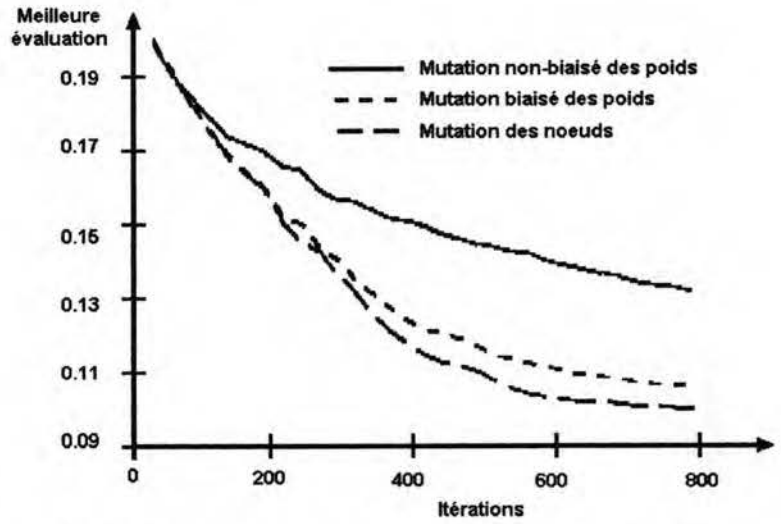


Figure 4-5 : Comparaison des trois types de mutations différentes

4.2.6.2.2.5 Mutation des nœuds les plus faibles ("Mutate-Weakest-Nodes")

Cet opérateur d'évolution repose sur la notion de *force* d'un nœud. La *force* d'un nœud, dans un réseau de neurones *feedforward*, est la différence entre la valeur d'évaluation avec le réseau intact et la valeur d'évaluation en désactivant toutes les sorties de ce nœud. Ces deux évaluations successives sont effectuées à l'aide d'un même vecteur d'entrée.

Cette mutation porte sur les nœuds cachés les plus faibles. La procédure suivante est exécutée:

- La force de chacun des neurones cachés est évaluée.
- Les m neurones les plus faibles sont sélectionnés.
- Si la force du nœud sélectionné est négative, une mutation non-biaisé est effectuée
- Si la force du nœud sélectionné est positive, une mutation biaisé est effectuée.

L'intuition de cet opérateur est que, certains nœuds sont sous utilisés (nœuds faibles). Une mutation sur ces nœuds est donc, en général, meilleure qu'une mutation sur un nœud choisi aléatoirement.

4.2.6.2.2.6 Comparaison entre la mutation classique et mutation des nœuds les plus faibles

En comparant les deux types de mutations : Mutation des nœuds et mutation des nœuds les plus faibles, voir figure 4.6, on n'obtient aucune nette supériorité.

La mutation des nœuds les plus faibles est meilleure au début qu'en fin d'opération. La notion de nœuds faibles proposée par ces deux chercheurs est donc à affiner.

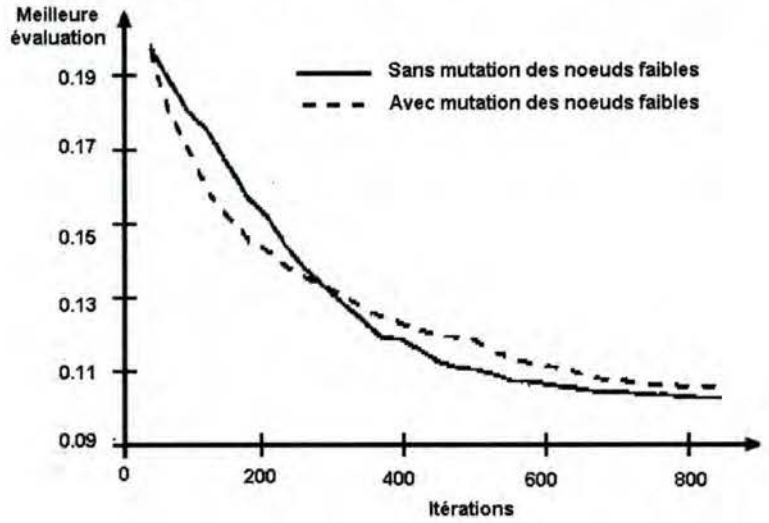


Figure 4-6 : Comparaison des deux types de mutations des nœuds

Cette comparaison a été effectuée en utilisant dans les deux cas les mêmes paramètres excepté le type de mutation. Pour l'opérateur d'évolution de croisement, le croisement des poids a été utilisé.

4.2.6.2.3 Le second opérateur d'évolution : Le croisement (*Crossover*)

4.2.6.2.3.1 Le croisement des poids (Crossover-Weight)

Deux parents sont choisis aléatoirement parmi la population des individus. Deux nouveaux vecteurs enfants sont créés au départ de ces deux parents. Pour chacune des différentes connexions des vecteurs enfants, les poids sont choisis aléatoirement chez l'un des deux parents.

4.2.6.2.3.2 Le croisement des nœuds ("Crossover-Nodes")

Deux parents sont choisis aléatoirement parmi la population des individus. Deux nouveaux vecteurs enfants sont créés au départ de ces deux parents. Pour chacun des différents nœuds des vecteurs enfants, on choisit aléatoirement un des deux parents et on copie les valeurs des poids de toutes les connexions entrantes du nœud correspondant au parent choisi.

4.2.6.2.3.3 Le croisement "poussé" ("Crossover-Features")

Dans les réseaux de neurones à couches entièrement connectées, le fait d'inverser un nœud A et un nœud B dans une même couche n'a aucune influence sur un nœud C de

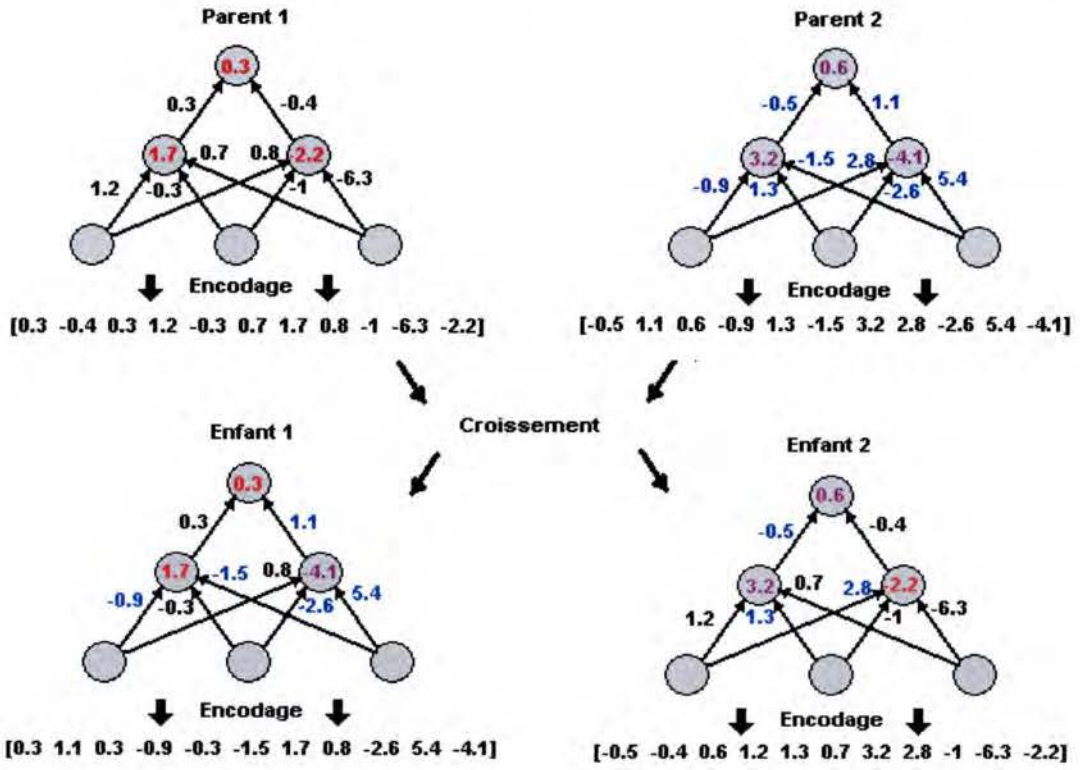


Figure 4-7 : Croisement des poids

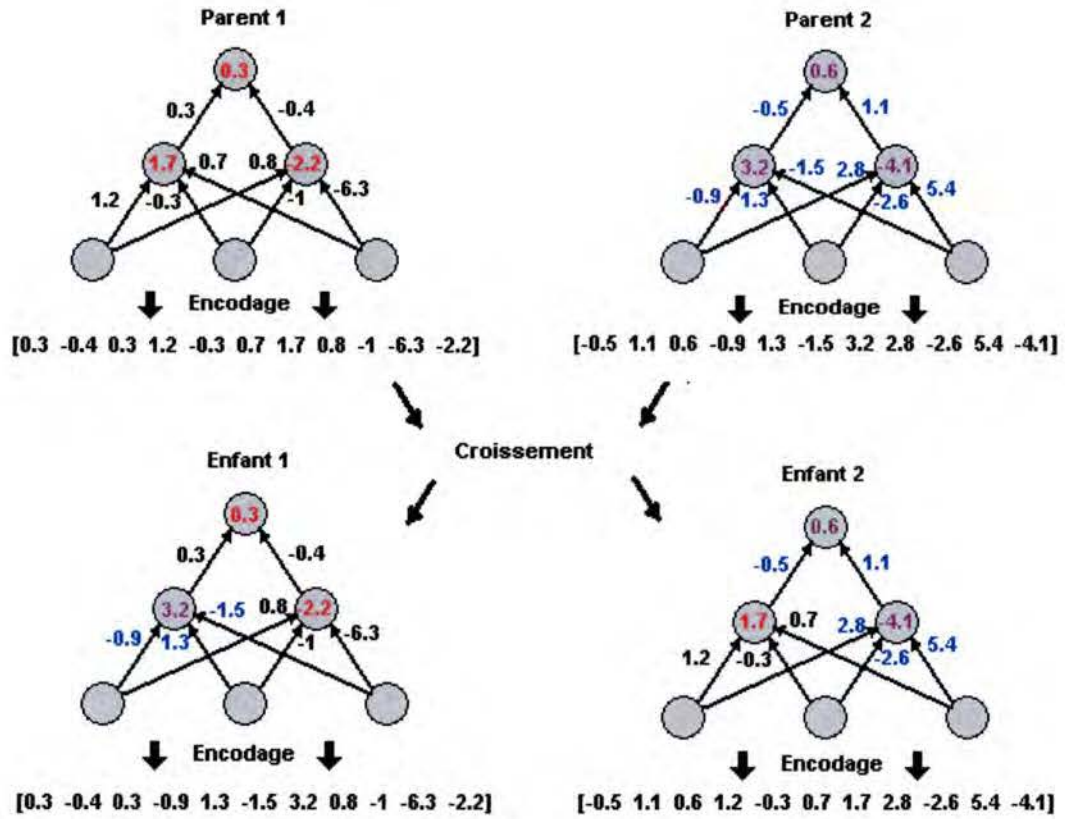


Figure 4-8: Croisement des nœuds

la couche suivante. Ce phénomène est la source du problème exposé au point 4.2.4, problème de la permutation (*Competing Conventions Problem).*

On peut supposer qu'effectuer l'opération de croisement sur deux nœuds similaires est plus bénéfique qu'effectuer cette opération sur deux nœuds pris aléatoirement. Avant d'effectuer l'opération de croisement, le second parent va donc subir un réarrangement pour que son architecture soit similaire au premier parent.

A chaque nœud du premier parent, on cherche un nœud dans le second parent qui joue le même rôle. Pour trouver ces nœuds similaires, une série d'entrées est présentée aux deux réseaux pris aléatoirement et une comparaison des réponses des différents nœuds est effectuée. On associe ainsi les nœuds qui sont les plus similaires. Une fois que cette opération est terminée pour tous les nœuds, l'algorithme de croisement des nœuds (vu au point précédent) est appliqué.

4.2.6.2.3.4 Comparaison des opérateurs de croisement

En comparant les trois types de croisement : le croisement des poids, le croisement des nœuds et le croisement poussé des nœuds, voir figure 4.9, des différences de performances très faibles avec une légère dominance du croisement poussé des nœuds (*Feature Crossover*) ont été obtenues. Il est cependant important de remarquer que l'augmentation du nombre de calculs à effectuer est non négligeable.

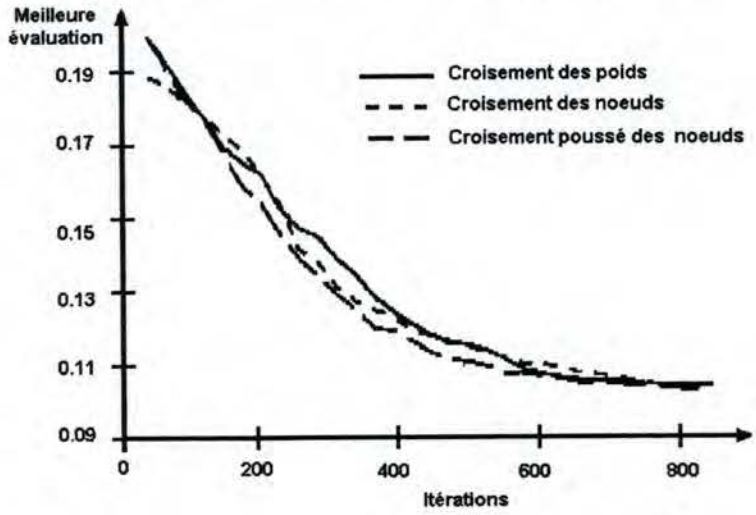


Figure 4-9:Comparaison des différents types de croisement

Cette comparaison a été effectuée en utilisant dans les trois cas, les mêmes paramètres excepté le type de croisement. Pour l'opérateur d'évolution de mutation, la mutation des nœuds a été utilisée.

4.2.6.2.4 Algorithme génétique

L'algorithme génétique proposé par Davis et Montana sur base de ces résultats, comporte donc les deux opérateurs d'évolution suivant : La mutation des nœuds et le croisement des nœuds.

4.2.7 Apprentissage hybride (algorithme génétique GA + rétropropagation du gradient BP)

4.2.7.1 Introduction

Un des problèmes principaux des GA algorithmes génétiques est leur inefficacité à converger rapidement localement ("problem of fine-tuned"). Cependant il présente l'avantage d'une bonne recherche globale sur un ensemble de solution.

Le problème des algorithmes de descente de gradient, comme la rétropropagation du gradient BP, est qu'ils convergent très difficilement dans le cas des réseaux de neurones complexes (fonction d'erreur à minimiser avec plusieurs optima locaux). Cependant, ils présentent l'avantage d'une convergence rapide vers un de leurs sous optima locaux.

4.2.7.2 Les algorithmes d'apprentissage hybrides GA + BP

Sur base des constatations du point précédent, R. Belew, J. McInerney et N. Schraudolph ont proposé une approche très intéressante pour l'apprentissage de réseaux de neurones en combinant les avantages des GA et de la BP [BMIS90].

Les GA sont une bonne méthode pour balayer tout l'espace des solutions et la BP permet une rapide convergence locale (ou d'autres procédures de descente de gradient), voir figure 4.10.

L'idée est donc d'utiliser les algorithmes génétiques pour obtenir et sélectionner de bons individus et d'utiliser la BP pour converger vers les sous-optima locaux voisins des points sélectionnés par l'GA.

L'optimum global est le minimum des optima locaux trouver par BP.

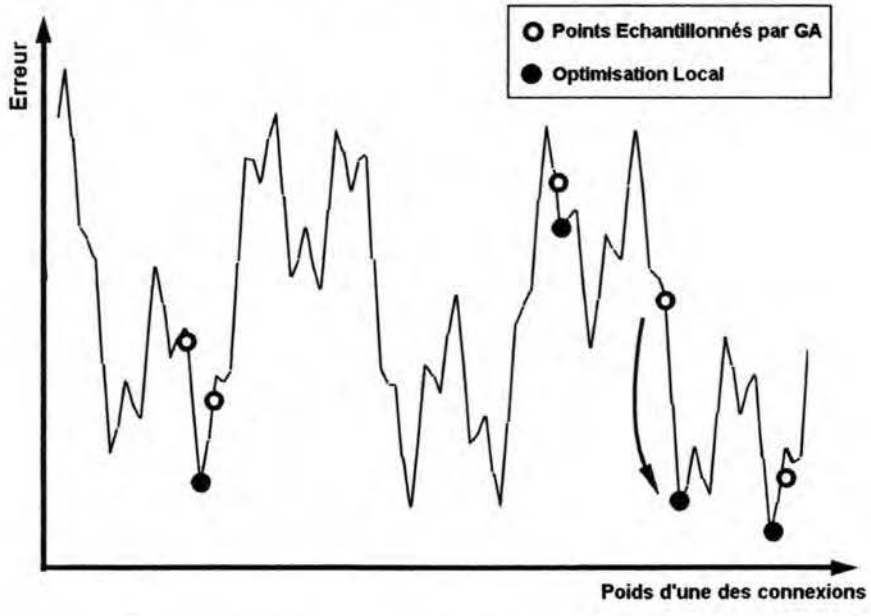


Figure 4-10 : Echantillonnage par GA et optimisation local par BP

La stratégie suggérée est donc d'utiliser les GA pour créer des semences ("seeds"), ces semences seront les points de départ de la procédure de recherche locale. Une vue schématique de la construction *hybride* est présentée à la figure 4.11.

- Le GA sélectionne les vecteurs de poids initiaux **Wi(0)** (W(0) est la matrice de tous les vecteurs de poids de tous les individus du groupe d'apprentissage)
- Pour chaque individu de la population, on effectue la BP pour un nombre défini d'itérations, on obtient les vecteurs des sous optima locaux **Wi***
- L'apprentissage des réseaux de neurones représenté par les différents vecteurs de poids **Wi*** est effectué. La fonction d'évaluation **μ=MSE(Wi*)** (c-à-d l'erreur au sens des moindres carrés *Minimum Squared Error*) est calculée pour chacun des individus du groupe d'apprentissage.
- La valeur de μ est renvoyée comme *fitness* aux différents individus du groupe d'apprentissage.

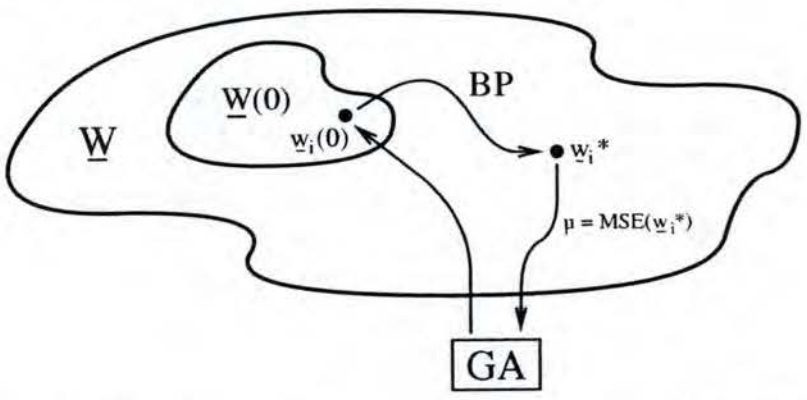


Figure 4-11: Estimation des fitness des différents individus W(0)

4.2.7.3 Condition d'initialisation

Il est important de remarquer que la convergence de la BP est très sensible aux conditions d'initialisation et que trouver cette zone d'initialisation n'est pas un problème trivial.

4.2.8 Comparaison de l'apprentissage par GA, BP et hybride (BP + GA)

4.2.8.1 Comparaison GA (algorithmes génétiques) et BP (rétropropagation du gradient)

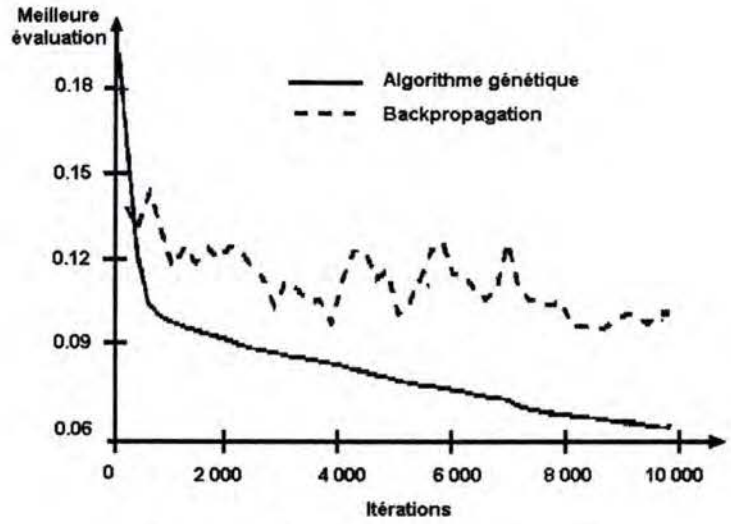


Figure 4-12 : Comparaison BP et GA

En comparant ces deux types d'apprentissage : Algorithme génétique GA et *rétropropagation* du gradient BP, voir figure 4.12, on voit clairement que l'apprentissage par GA surpasse l'apprentissage par BP dans l'application étudiée par Davis et Montana (BP classique utilisé par Rumelhart [Rum86] avec un taux d'apprentissage de 0.5.). Cette constatation a tendance à se généraliser lorsque l'apprentissage porte sur des réseaux de neurones complexes. D'autres études ont été effectuées par S. Sexton, R. Dorsey et J. Johnson en 98 ont confirmé ces résultats pour d'autres applications [SDJ98].

4.2.8.2 Comparaison BP, GA et méthode hybride (GA + BP)

Pour effectuer cette comparaison il faut séparer les applications en deux grandes catégories en fonction de la complexité du réseau de neurones sur lesquels porte l'opération d'apprentissage.

- Les réseaux de neurones simples, avec un seul optimum pour la fonction d'erreur à minimiser. Pour ce type d'application la BP présente l'avantage de converger rapidement et efficacement vers l'optimum global. Les GA et la méthode hybride convergeront également mais sont plus gourmands en temps calculs
- Les réseaux de neurones complexes avec plusieurs sous optima locaux pour la fonction d'erreur à minimiser. Pour ce type d'application la tendance générale qui tend à se dégager est représentée à la figure 4.13. L'algorithme le plus performant est l'algorithme *hybride*, il est meilleur que l'algorithme génétique. L'algorithme de BP converge rapidement vers un des sous optima locaux, avec le temps il peut trouver d'autres sous optima locaux, meilleurs, mais rarement l'optimum global.

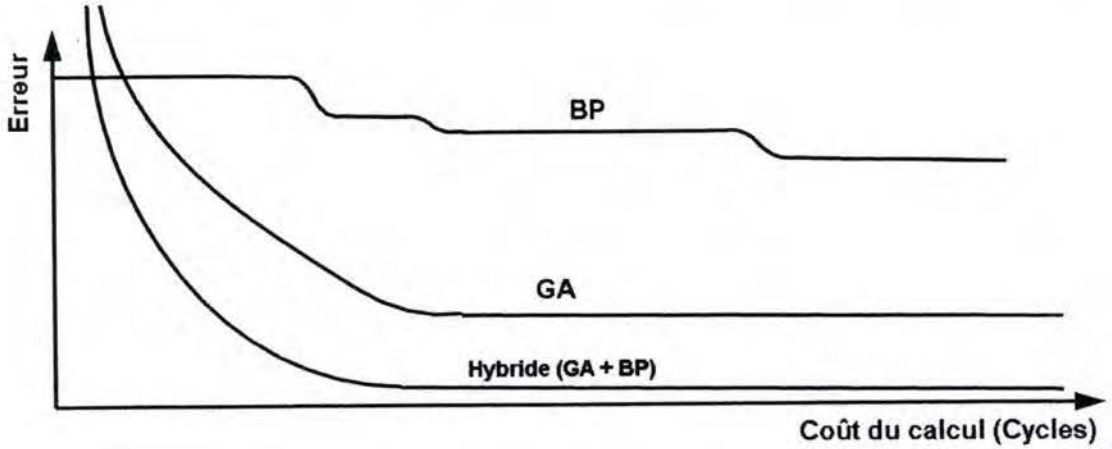


Figure 4-13 : Comparaison BP, GA et GA + BP

Belew a utilisé cette technique dans différentes applications et il a obtenu des résultats très concluants [BeMISc90].

4.2.9 Conclusion

L'apprentissage des poids d'un réseau de neurones, par algorithme génétique et par algorithme hybride est donc plus performant que l'apprentissage classique par BP, pour des applications de réseaux de neurones complexes (c-à-d lorsque la fonction à minimiser est multimodale). Ce couplage des algorithmes génétiques aux réseaux de neurones est donc une première application intéressante.

4.3 Architecture optimale de réseaux de neurones par algorithmes génétiques

4.3.1 Introduction

Dans la plupart des applications des réseaux de neurones, l'architecture du réseau, c'est-à-dire le nombre d'unités ainsi que leurs connexions, est fixée au départ. Ce choix est souvent dicté par des heuristiques qualitatives du type : "le nombre d'unités cachées augmente avec la difficulté du problème" et par essais erreurs. Les chercheurs en réseau de neurones savent bien que le choix de l'architecture du réseau peut conduire à la réussite ou à l'échec d'une application, c'est pourquoi un des buts visés par beaucoup de ces chercheurs, est l'optimisation automatique de l'architecture du réseau pour une application particulière. Beaucoup pensent que les algorithmes génétiques peuvent être très utilisés pour cette tâche. Beaucoup de recherches ont été effectuées dans ce domaine. Afin d'obtenir des croisements et des mutations efficaces, ces opérations sont effectuées sur les **génotypes** (codage du réseau de neurones), plutôt que sur les **phénotypes** (le réseau lui-même). Les étapes clés de ce processus d'optimisation d'architecture sont représentées dans le schéma ci-dessous.

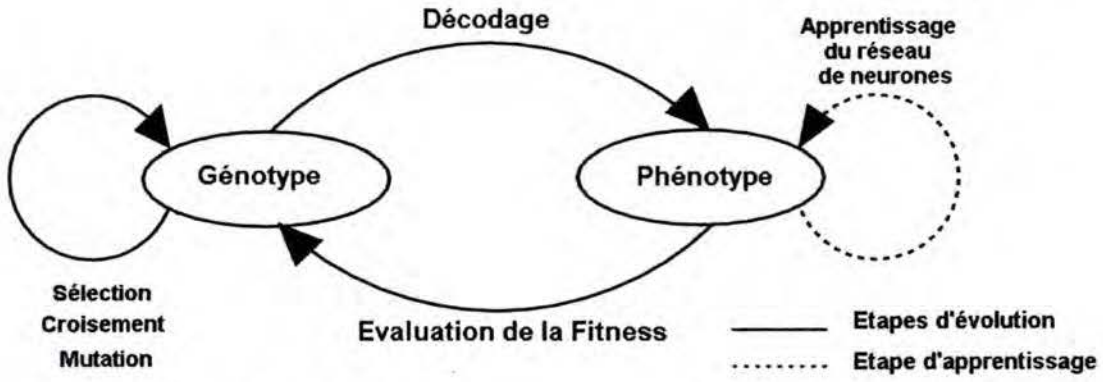


Figure 4-14 : Schéma des différentes étapes pour l'apprentissage de réseau de neurones par algorithmes génétiques

Les résultats des recherches peuvent être répartis en quatre grandes catégories différentes. Ces catégories sont fonctions du type d'encodage utilisé.

- **L'encodage direct**: Le réseau est directement encodé dans un chromosome. Cet encodage requiert très peu d'effort à la fois pour l'encodage ainsi que pour le décodage des chromosomes. La transformation du génotype en phénotype est triviale. Un exemple de cet encodage est la matrice de connexions qui spécifie précisément et directement l'architecture du réseau correspondant.
- **L'encodage paramétrique:** Le réseau est encodé à l'aide de paramètres. Cette méthode diminue la longueur des chromosomes dans le cas de réseaux de grande taille.

- **L'encodage de grammaire**: Une série de règles est encodée, ces règles seront utilisées pour développer le réseau de neurones. Ce type d'encodage demande un effort considérable pour passer des génotypes aux phénotypes.
- **L'encodage par programmation génétique:** Cet encodage est effectué à l'aide d'un paradigme particulier de programmation.

4.3.2 Encodage direct

La méthode d'encodage direct a été illustrée dans les travaux de Geoffrey Miller, Peter Todd et Shailesh Hegde en 1989 [Mil89]. Leurs travaux ont été focalisés sur les réseaux feedforward, avec un nombre fixe de neurones. L'algorithme génétique est utilisé pour trouver les connexions optimales entre ces neurones.

4.3.2.1 Codage du réseau

Comme montrée dans la figure ci-dessous, la topologie du réseau est représentée par une matrice de carrés de dimension N (N égale 5 dans l'exemple ci-dessous). Dans la matrice la valeur 1 dans la ligne *i* et colonne *j* représente une connexion allant du neurone *i* vers le neurone *j*, la valeur 0 indique l'absence de liaison entre les deux neurones.

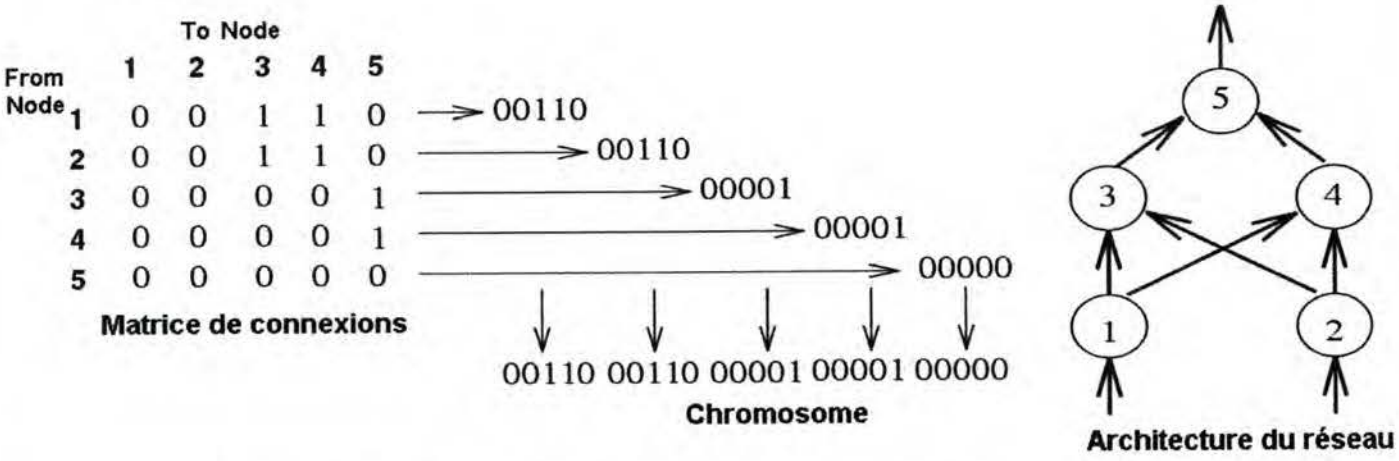


Figure 4-15 : Méthode d'encodage direct pour un réseau de neurones

4.3.2.2 Sélection

Miller, Todd et Hegde utilisent la sélection proportionnelle aux fitness.

4.3.2.3 Croisement

Pour l'opération de croisement, un numéro de colonne est choisi aléatoirement et les colonnes correspondantes de 2 parents sont échangées. Ce croisement est similaire au croisement des nœuds de Davis et Montana, car une colonne représente toutes les connexions entrantes dans un nœud.

4.3.2.4 Mutation

La mutation utilisée par ces chercheurs inverse aléatoirement la valeur de bits de la matrice de connexions.

4.3.2.5 Calcul des Fitness

La fitness d'un chromosome est obtenue en utilisant la méthode de rétropropagation du gradient BP, qui va ajuster les poids à l'aide des éléments d'un groupe d'apprentissage. Une fois la convergence obtenue, la fitness est la somme des moindres carrés entre les valeurs attendues à la sortie du réseau et les valeurs réelement obtenues.

4.3.2.6 Applications

Ces trois chercheurs ont utilisé les algorithmes génétiques dans trois applications différentes.

- L'optimisation d'architecture pour le problème du XOR (décrit plus haut).
- L'optimisation d'architecture pour le problème des 4 quadrants: les valeurs d'activation réelles (entre 0 et 1) des 2 unités d'entrée représentent les coordonnées d'un point dans un carré. Toutes les entrées présentes dans le quadrant inférieur gauche et dans le quadrant supérieur droit produisent une activation sur l'unique neurone de sortie égale à 1 et tous les autres points produisent une activation égale à 0.
- L'optimisation d'architecture pour le problème d'encodage/décodage 4-X-4 (4 entrées, 4 sorties et X neurones cachés). Les unités de sortie doivent copier l'individu initial présenté aux unités d'entrée. Ce problème peut sembler trivial et pour qu'il présente un intérêt, le nombre de couches cachées doit être inférieur au nombre de couches d'entrée (un encodage et un décodage sont donc réalisés).

Ces problèmes sont relativement faciles pour un apprentissage par la méthode de rétropropagation du gradient. Pour chaque application :

- taille de la population est de 50,
- taux de croisement est de 0.6,
- taux de mutation est de 0.005.

Pour ces 3 applications, les algorithmes génétiques ont trouvé facilement une topologie présentant une faible erreur. Cependant ces problèmes ne sont pas trop compliqués. Par contre, pour des applications demandant des réseaux de tailles plus importantes, ce modèle a des limites.

L'encodage direct proposé par ces chercheurs n'est qu'une possibilité d'encodage parmi beaucoup d'autres, par exemple, au lieu d'encoder simplement l'existence de

connexions, on peut encoder les poids et les biais des connexions. D'autres applications plus complexes ont été présentées dans l'ouvrage de Whitley et Schaffer [Sch92].

4.3.3 Encodage paramétrique

A Harp, T. Samad, A. Guha ont proposé une méthode d'encodage compacte pour des réseaux de neurones avec architecture en couches complètement connectées [Harp89] et [Harp90]. Cet encodage s'effectue à l'aide de paramètres. Cette méthode encode uniquement la structure du réseau, sans encoder les poids des connexions. La caractérisation de la structure doit inclure le nombre de neurones, le nombre de couches, et une mesure de la connectivité de chacun des neurones. Ajouter un neurone modifie simplement un seul paramètre dans le chromosome plutôt que d'augmenter la taille du chromosome comme dans l'encodage direct. Le chromosome peut également inclure dans sa structure les paramètres de l'apprentissage du réseau, comme le taux d'apprentissage η et momentum α pour le rétropropagation du gradient BP.

Dans la méthode Harp, le génome est divisé en différentes parties que l'on appelle *aires*. Chaque *aire* représente une couche. La première *aire* est la couche d'entrée, la dernière *aire* est la couche de sortie. Chaque aire est divisée en deux parties. La première est la partie qui encode les paramètres internes de l'aire : le numéro identifiant l'aire, le nombre de neurones dans cette couche, les coordonnées dans l'espace de l'aire. La seconde partie encode les différentes projections. Ces projections sont les connexions vers les autres aires. Pour chaque projection les paramètres encodés sont : les coordonnées de l'aire de destination ainsi que la densité de cette connexion.

L'opérateur principal d'évolution est le croisement. Cependant pour avoir de bonnes propriétés de mise à l'échelle, la mutation est également présente, ce qui permet l'ajout et la suppression d'aire et de projection.

Cette méthode est plus abstraite que l'encodage direct mais permet d'encoder de très larges réseaux avec des chromosomes de petite taille. Sa seule restriction est son utilisation uniquement pour des réseaux de neurones avec architecture en couches complètement connectées.

4.3.4 Encodage de grammaire

4.3.4.1 Introduction aux fractales et aux L-systèmes

L'encodage de grammaire repose sur la théorie des fractales et des *L-systèmes.* Pour construire des réseaux de neurones complexes, les chercheurs se sont inspirés (une fois de plus) de ce qui se passe dans les systèmes biologiques, à savoir le développement et la croissance d'organismes vivants. Chaque individu vivant contient des informations génétiques, les génotypes, qui déterminent la forme finale de l'organisme développé, le phénotype. Cette construction du phénotype au départ du génotype, n'est pas directe dans les systèmes biologiques. On ne construit pas avec des méthodes du type "papier carbone", mais selon des méthodes utilisant des recettes indirectes de construction. Le processus utilisant ces "recettes de cuisine" est appelé **l'ontogenesis** d'un organisme, c'est en fait, le long et complexe processus de division et de spécialisation de cellules, suivant des règles encodées dans le génome qui permettent de donner un individu adulte.

Beaucoup de méthodes d'évolution de réseau de neurones sont inspirées *des systèmes de Lindenmayer* (L-système). Le formalisme et les applications de cette idée ont été introduits par Lindenmayer en 1976. A l'origine cette approche grammaticale a été utilisée pour modéliser la morphologie des plantes au départ de règles utilisant des caractères. Ce processus est appelé *réécriture de caractères.* Cette réécriture de caractère commence par un caractère initial, *l'axiom.* Chaque membre de gauche d'une règle trouvée est remplacé par son membre de droite. Par exemple si nous avons un caractère initial F et que nous avons la règle suivante

$$F \rightarrow F\text{-}F{+}{+}F\text{-}F.$$

Ce caractère est remplacé par "F-F++F-F". En appliquant cette règle une seconde fois on obtient "F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F". Cette règle est appliquée un certain nombre de fois prédéfini à l'avance. Ces symboles {F,-,+} peuvent être utilisés comme les mouvements d'une tortue sur une surface (cf le déplacement de la tortue utilisée dans le logiciel *Logo*). F représente l'avancement de la tortue d'un pas déterminé, - signifie que la tortue doit effectuer une rotation à gauche d'un angle δ et + une rotation à droite d'un angle δ. En prenant δ = 60°, si un marqueur est attaché à la tortue, la figure de gauche ci-dessous est dessinée. C'est la première fractale introduit par Koch en 1904.

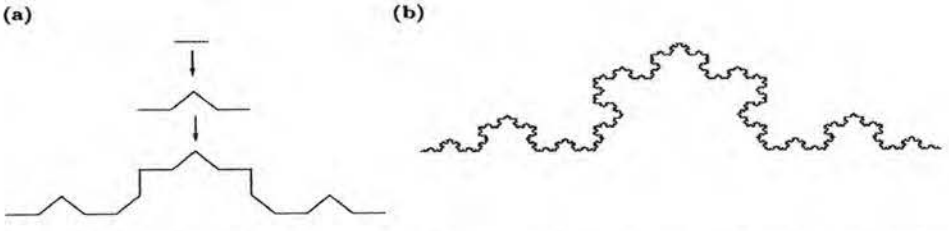


Figure 4-16 : Fractale obtenue par substitution de la règle "F->F-F++F-F"

Le modèle de Koch n'est pas très utilisé pour modéliser des morphologies. En introduisant deux nouveaux symboles "[" et "]"on peut avoir des dessins plus proches d'organismes vivants. Le premier symbole "[" stocke au sommet d'une pile la position et la direction de la tortue. Le second symbole "]" restaure la position de la tortue stockée au sommet de la pile. Ces symboles donnent la possibilité à la tortue de se déplacer. On peut voir à la figure ci-dessous avec d= 23° et 5 itérations que l'organisme modélise par la grammaire suivante, ressemble beaucoup plus à une plante.

$$X \rightarrow F\text{-}[\text{-}[[X]]\text{+}X]\text{+}F[\text{+}FX]\text{-}X]$$

$$F \rightarrow FF$$

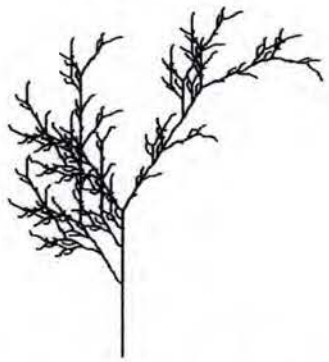


Figure 4-17 : Fractale obtenue par substitution des règles
$X \rightarrow F\text{-}[\text{-}[[X]]\text{+}X]\text{+}F[\text{+}FX]\text{-}X]$ et F->FF

On peut trouver également, des *L-systèmes* avec sensibilité de contexte. Le contexte peut être à gauche *"<"*, à droite *">"* ou des deux côtés *"< et >"* d'un groupe de caractère. Voici la forme générale de règle dans un *L-système* sensible au contexte.

$$L < P > R \rightarrow S$$

P est appelé prédécesseur et S successeur, respectivement membre de gauche et membre de droite de la règle de production. L et R contexte de gauche et contexte de droite peuvent être présents ou absents. En termes techniques, une grammaire sans règle de contexte est appelée *0L-système*, s'il y a du contexte d'un seul côté elle est appelée *1L-système* et si le contexte est présent des 2 côtés, *2L-système*. Une règle de production avec des contextes L et R à gauche et à droite ne peut être remplacée par S uniquement si P est précédée de L et suivi de R. Si deux règles de production peuvent être utilisées pour remplacer un certain caractère, l'une avec contexte, l'autre sans contexte, celle avec contexte sera utilisée.

Cette modélisation de *Lindenmayer* crée des structures proches de plantes, mais il ne dit pas que les plantes se construisent de cette manière, c'est juste une modélisation de morphologie d'organisme. En fait, comme on le sait depuis longtemps, les arbres grandissent par divisions de cellules. L'utilisation de système de réécriture de caractère est juste une des tentatives idéalisées de modélisation de ce processus.

4.3.4.2 Méthode de Kitano

En 1990, Hiroaki Kitano publie une étude qui met en évidence que la méthode d'encodage directe devient de plus en plus difficile lorsque la taille du problème augmente [Kita90]. Lorsque le nombre N de neurones du réseau augmente, la taille du chromosome utilisé pour encoder ce réseau augmente en N^2, ce qui conduit à la fois à des problèmes de performance et d'efficacité. En fait, certaines sous-structures d'un réseau présentent plusieurs fois peuvent être représentées une seule fois, par un seul et même symbole. Sur base de cette observation Kitano a proposé dans son étude l'encodage du réseau à l'aide de règles grammaticales utilisant des symboles. Ces symboles représentent des parties du réseau. L'algorithme génétique qu'il a présenté effectue des opérations de croisement et de mutation sur les génotypes et les fitness ne sont calculées qu'après avoir effectué les différentes étapes de développement du réseau à l'aide de ces règles grammaticales.

4.3.4.2.1 Encodage

Cette idée de Kitano a été appelée par la suite "grammaire de génération de graphe". Un exemple simple de ce type de grammaire est donné dans la figure ci-dessous. Le membre de droite de chaque règle est une matrice de dimension 2x2, le membre de gauche est un caractère de dimension 1. Les lettres majuscules sont des "non-terminaux", et les lettres minuscules sont des terminaux allant de *a* à *p* et représentent les 16 matrices possibles 2x2 de 1 ou de 0. Chaque non-terminal a exactement un et un seul membre de droite. Il n'y a donc qu'une seule structure possible, au départ de la grammaire du point A de la figure ci-dessous pour obtenir le réseau du point D.

Dans les expériences de Kitano, la valeur 1 en ligne *i* et en colonne *i* indique que le neurone est présent dans le réseau. La valeur 1 dans la ligne *i* et la colonne *j* avec $i \neq j$, indique qu'il y a une connexion de l'unité *i* vers l'unité *j*. S'il y a une connexion de ou vers un nœud n'existant pas elles sont ignorées, tout comme les connexions récurrentes. Dans la figure ci-dessous un réseau de neurones modélisant la fonction booléenne XOR est représenté.

Le chromosome représenté au point A de la figure ci-dessous est constitué de plusieurs groupes comprenant 5 caractères. Le premier caractère représente le membre de gauche d'une règle grammaticale et les 4 autres caractères représentent le membre de droite.

Le premier groupe de 5 caractères est différent des suivants, il commence toujours par le symbole de départ S et est suivi de caractères choisis aléatoirement entre A et Z (symbole non terminal).

Les groupes suivants sont constitués comme suit, le premier caractère est compris entre A et Z, les quatre autres caractères sont compris entre *a* à *p* (symbole terminal).

Si dans des groupes différents on trouve pour le premier caractère plusieurs fois la même valeur, la règle choisie est celle obtenue par le premier de ces groupes.

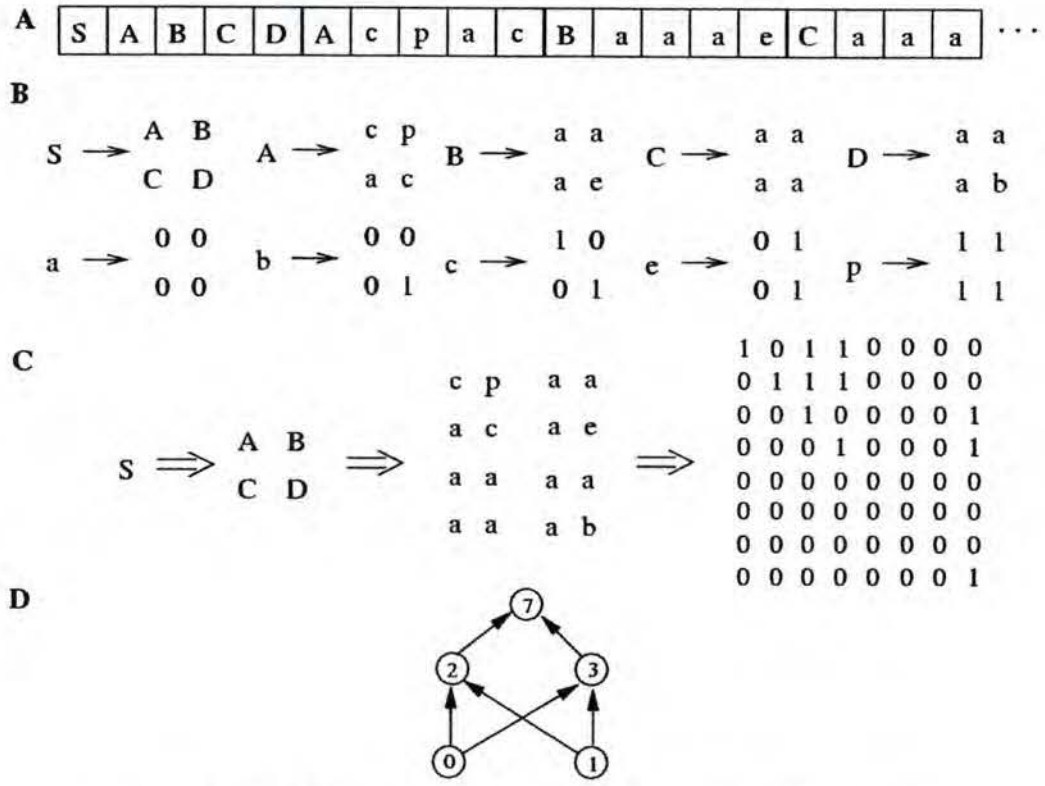


Figure 4-18 : Les différentes étapes de l'algorithme de Kitano

4.3.4.2.2 Sélection

L'algorithme génétique utilise également la sélection proportionnelle aux valeurs des fitness.

4.3.4.2.3 Croisement

Le croisement multi-points est utilisé.

4.3.4.2.4 Mutation

La mutation utilisée est la mutation adaptative, et ici, on remplace aléatoirement certains caractères par une des valeurs correspondantes soit entre A et Z ou soit entre *a* et *p*.

4.3.4.2.5 Fitness

Une fois le réseau développé, la méthode d'obtention de la fitness est identique à celle proposée pour l'encodage direct par Miller, Todd et Hegde.

4.3.4.2.6 Applications

Kitano a effectué une comparaison entre sa méthode et la méthode d'encodage direct. Cette comparaison a été effectuée sur le problème de l' "encodage/décodage" de type 4-x-4 et 8-x-8. Sur ces applications relativement simples, il a montré que sa méthode surpasse l'encodage direct et ce d'autant mieux que la taille du réseau augmente. Les performances décroissent très vite avec la taille du réseau pour l'encodage direct, par contre celles-ci restent relativement constantes pour la méthode de Kitano.

Il a également fait varier le nombre de règles de production encodées dans le génotype de 5, 10, 15, 20 et 40. Cela lui permit de tirer la conclusion suivante : Plus le nombre de règles encodées est grand, meilleur est le résultat en fin d'évolution.

Kitano indique que les méthodes utilisant un encodage de grammaire doivent normalement surpasser l'encodage direct, car elles permettent de créer des groupes répétitifs de connexions régulières (ces structures découlent naturellement de la répétition de règles grammaticales) ce qui est bénéfique.

Une extension du travail de Kitano est présentée dans [Kita94] où on effectue en même temps que l'optimisation de l'architecture du réseau, l'optimisation des poids des connexions. Ces poids obtenus sont les points de départ pour l'apprentissage de la rétropropagation du gradient.

4.3.4.3 Boers et Kuiper

Boers et Kuiper ont également proposé, comme Kitano, dans leur étude publiée en 1992 [Boe92], une modélisation de réseau de neurones utilisant une grammaire *L-système mais cette fois avec contexte.* Cette grammaire modélise des neurones ainsi que des groupes de neurones.

4.3.4.3.1 Encodage

Les neurones sont marqués avec un symbole entre A et H. Les modules sont notés entre crochets "[ABAG]" et sont manipulés de la même manière que les neurones individuels. Chaque neurone ou groupe de neurones est connecté par défaut au neurone ou au groupe de neurones suivant, "AB" signifie que A est connecté à B. Les connexions manquantes sont notées par une virgule, comme par exemple "A,B".

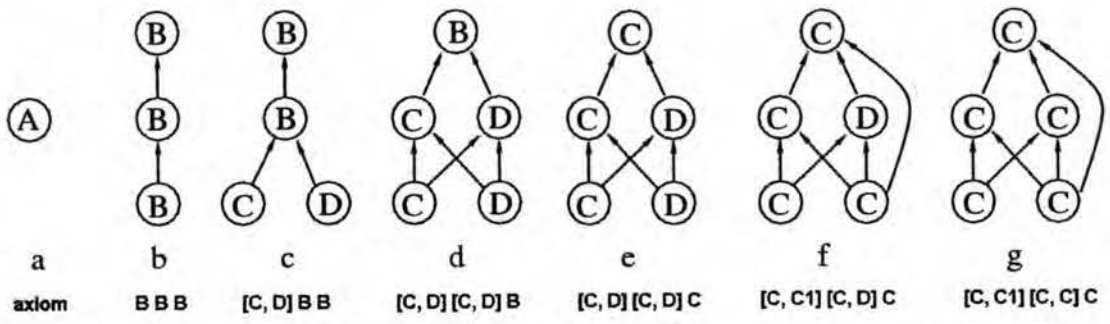


Figure 4-19: Transformation du neurone initial A vers réseau final "[C,C1][C,C]C"

Les connexions supplémentaires sont notées par un chiffre qui indique combien de neurones la connexion passe. Par exemple: "A3AAAA" décrit une chaîne de 4 neurones avec le premier neurone A connecté au 4ème neurone.

Un exemple est présenté à la figure suivante, on passe de la configuration A à la configuration G en utilisant les règles décrites à la figure 4.21.

1.			A			→	BBB
2.			B	>	B	→	[C,D]
3.			B			→	C
4.	C	<	D			→	C
5.			D	>	D	→	C1

Figure 4-20 : règle de transformation du réseau

Les règles sont des règles contextuelles. Pour la règle 4, un nœud D n'est transformé en C, uniquement que s'il est précédé d'un nœud C. Pour la règle 5, un nœud D n'est transformé en C1, uniquement que s'il est suivi d'un autre nœud D.

Les différentes règles sont encodées les unes à la suite des autres dans un chromosome. Les éléments (L, P, R ou S) de ces règles de production sont séparés lors de l'encodage par le symbole "*". Pour encoder un de ces 17 caractères dans un chromosome qui sera manipulé par un algorithme génétique, les chercheurs ont utilisé un codage spécial. Le codage de ces 17 caractères a été effectué avec 64 chaînes de 6 bits, par analogie au codage des 20 acides-aminés sur 64 triplets avec 4 bases. La table de codage est représentée à la figure ci-dessous.

	00	01	10	11	
00	3	[	D	]	00
	3	[	D	]	01
	*	[	2	2	10
	*	[	2	5	11
01	*	1	E	]	00
	*	1	E	]	01
	*	1	F	]	10
	*	1	F	]	11
10	2	A	G	[	00
	2	A	G	[	01
	2	A	H	]	10
	4	A	H	]	11
11	.	B	*	C	00
	.	B	*	C	01
	.	B	[	C	10
	.	B	[	C	11

Figure 4-21 : La table de codage utilisée par Boers et Kuiper par exemple la chaîne 100100 est le premier A de la table.

Description de l'algorithme de décodage:

1. Prendre un point de départ n'importe où dans la chaîne de bits. Démarrer la lecture de la partie L de la règle de production.
2. Lire les bits, 6 à la fois, regarder quel est le caractère correspondant dans la table ci-dessus.
3. Si ce caractère n'est pas un astérisque, ajouter le caractère au caractère déjà obtenu
4. Dans le cas contraire, assigner les caractères lus à la partie correspondante de la règle (L, P, R ou S). Avancer à la partie suivante de la règle de production. Si la dernière partie lue était S, commencer la lecture d'une nouvelle règle de production.
5. Répéter les étapes de 2 à 4, jusqu'à ce qu'on arrive à la fin du chromosome. S'il n'y a plus de bit à lire et que la règle n'est pas terminée, supprimer cette règle.

Il est possible de décoder le chromosome de 12 manières différentes en fonction de l'endroit où on commence la lecture et du sens de lecture. Il y a 6 possibilités dans le sens gauche-droite et les 6 autres possibilités dans le sens droite-gauche. Ces différents types de décodage induisent un très haut niveau de parallélisme par rapport aux applications classiques de codage. La figure ci-dessous est un exemple qui montre la traduction d'un chromosome en utilisant 4 des 12 possibilités.

On obtient les 4 chaînes suivantes: ***2]\[[]A* ,H[2[D, *A*BBB*** et **,*2]]C1,**

La troisième chaîne de caractères est par exemple une partie de la chaîne suivante : ...****A*BBB*C*...** qui correspond en fait à la règle suivante : A > BBB -> C (qui est la 1ère règle de la figure 4-20)

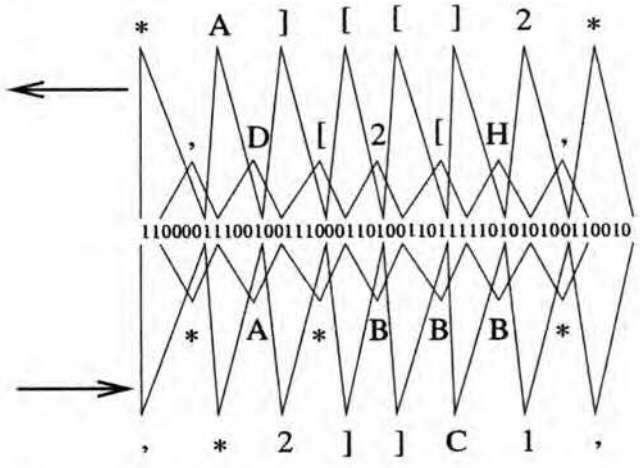


Figure 4-22 : Extraction du chromosome en 4 des 12 translations possibles

4.3.4.3.2 Opérateur de l'algorithme génétique

Les opérations classiques de mutation, de croisement et de sélection sont effectuées sur ces chaînes de bits.

4.3.4.4 Nolfi et Parisi

Nolfi et Parisi ont proposé un modèle inspiré des *L-système* fractales exposés ci-dessus. Le développement des axones entre les neurones est régi par des règles de fractale. Ce développement est itératif, les règles de croissance sont appliquées un certain nombre de fois défini à l'avance. Les règles sont de la forme suivante :

$$\mathbf{X} \rightarrow \mathbf{F}\,[\,-\mathbf{X}\,]\,[\,+\mathbf{X}\,] \quad \text{Pour les 4 premières itérations}$$

$$\mathbf{X} \rightarrow \mathbf{F} \quad \text{Pour la dernière itération}$$

"F" est la longueur de chaque segment, *"-"* et *"+"* sont les rotations à gauche et à droite sur base de l'angle de séparation, cet angle est identique pour les différentes itérations successives (voir figure 4.23.).

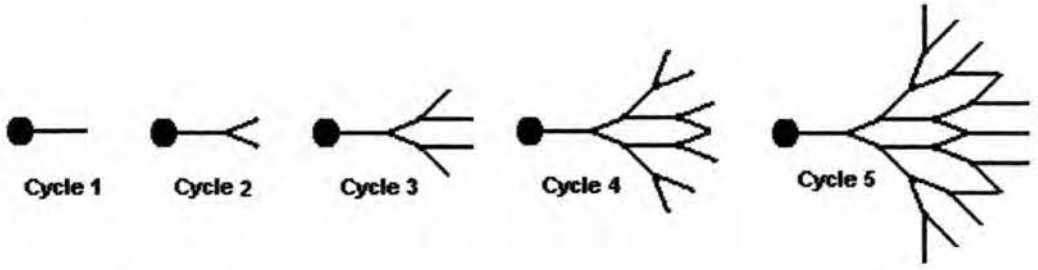


Figure 4-23 : Les 5 cycles de croissance d'un axone au départ d'un neurone

Les génotypes de chaque neurone sont constitués des paramètres suivants :

Nom des gènes	Type des gènes	Longueurs (bits)	Minimum	Maximum
expression	boolean	1	-	-
coord.x	bit-float	5	0	8
coord.y	bit-float	5	0	20
biais	bit-float	10	-1	1
poids	bit-float	10	-1	1
longueur segment	bit-float	4	0	1
longueur angle	bit-float	6	-1	1
type	bit-int	4	0	15

Figure 4-24 : Représentation génomique pour l'encodage de Nolfi et Parisi

Les neurones possèdent des coordonnées x et y pour pouvoir être placés dans un espace à deux dimensions (voir figure 4-25). Cet espace est séparé en trois parties verticales, les neurones se situant à gauche sont des neurones d'entrée, les neurones se situant dans la zone de droite, sont des neurones de sortie et les neurones de la zone centrale sont des neurones cachés. La transformation des gènes en neurones est directe mais les connexions entre ces neurones sont obtenues par la règle fractale de croissance d'axones. Le processus de croissance d'axones nécessite les valeurs des deux paramètres décrits ci-dessus, la longueur des segments et l'angle de séparation des segments. La croissance d'axone s'effectue en cinq cycles comme décrit à la figure 4.23 et on obtient une structure du type de la figure 4.26

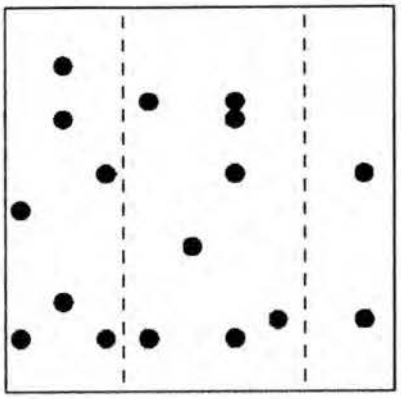


Figure 4-25 : Positionnement des neurones dans l'espace avec les coord.x et coord.y

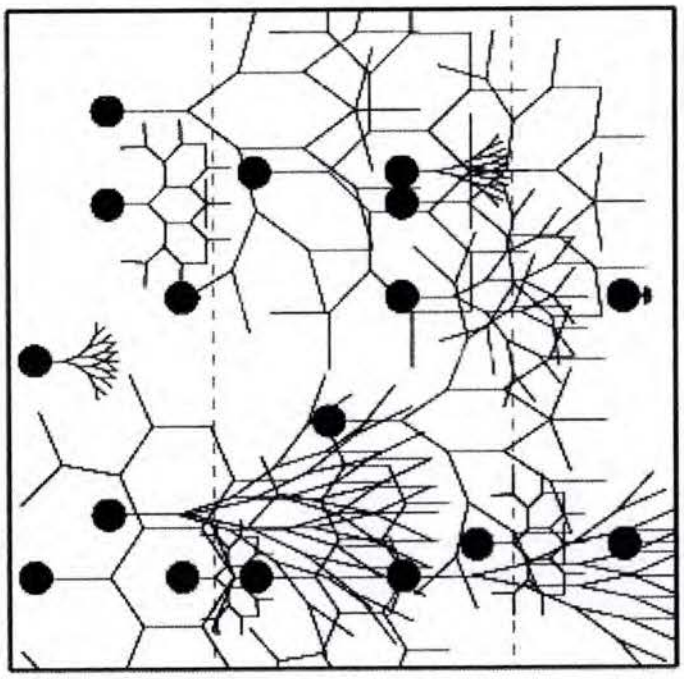


Figure 4-26: Phase de croissance des axones

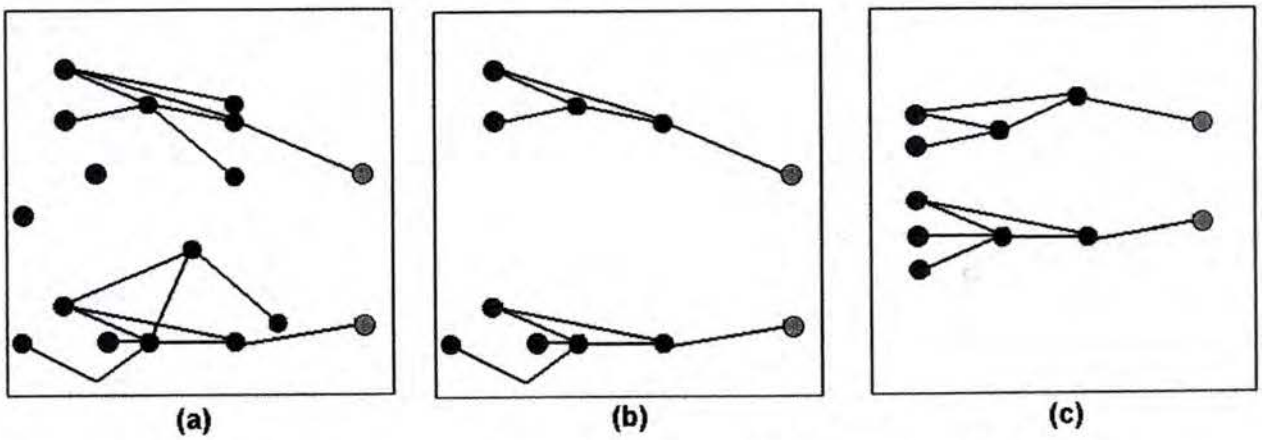


Figure 4-27: (a) Création des liens entre neurones (b) Suppression des neurones inutiles. (c) Réarrangement du réseau

Les poids ainsi que les biais sont également encodés dans le génome mais ne sont pas utilisés si un apprentissage est effectué après le développement du réseau.

Si la distance entre les axones et les autres neurones est inférieure à r_c (rayon de connectivité) une connexion est établie entre le neurone à l'origine de l'axone et le neurone suffisamment proche. Le poids de cette connexion est égal au poids du génome du neurone source. Les valeurs couramment utilisées pour r_c sont comprises entre 0.5 et 2. Les connexions feedback sont éliminées. Voir figure 4.27 (a).

Une fois les connexions établies, les neurones inutiles sont supprimés. On entend par neurone inutile, les neurones qui ne sont pas sur un chemin entre un neurone d'entrée et un neurone de sortie (voir figure 4.27 (b)). Et on obtient le réseau final (voir figure 4.27 (c)) et l'apprentissage peut commencer.

4.3.4.4.1 Opérateur de l'algorithme génétique

La sélection, le croisement et la mutation classique de chaînes de bits sont utilisés.

4.3.4.5 Cangelosi, Parisi et Nolfi

Cangelosi, Parisi et Nolfi ont proposé un modèle qui étend le modèle du point précédent [CPN95].

4.3.4.5.1 Encodage

L'architecture du réseau est également bidimensionnelle mais se développe au départ d'une cellule unique. La croissance de cette cellule s'étale sur 5 cycles de divisions et migrations dictées par des règles. Après cette phase de croissance de cellules, se déroule à nouveau l'étape de croissance des axones. Cette étape s'effectue également en 5 cycles de la même manière que la méthode précédente. La seule grande différence par rapport à la méthode précédente est la direction de croissance des axones qui n'est plus uniquement dans la direction de la gauche vers la droite, mais dans une direction quelconque dictée par la valeur du paramètre *"face"*.

Chaque individu commence donc avec une cellule unique. La cellule initiale contient trois sortes d'informations différentes codées dans le génome de la manière suivante.

1. Le type de cellules (il y a 16 types différents de cellules).
2. Différents paramètres concernant la croissance du neurone
    - L'angle de branchement de l'axone des neurones (il y a, à nouveau cinq cycles de branchements successifs, chaque branchement est une séparation d'une branche en deux sous-branches, tous les branchements pour un même neurone ont le même angle de branchement).
    - La longueur des segments de branchement de l'axone des neurones
    - La "face": La direction ou l'axone va commencer sa croissance.

- La valeur du biais (offset) de la fonction
- Le poids de la connexion émanant du neurone source (toutes les connexions émanent d'un même neurone ont le même poids)

3. Un ensemble de 16 règles de reproduction de cellules. Chaque règle est de la forme : **Type N = Type N' + Type N''**. Cette règle signifie que la cellule de type N se reproduit en deux cellules filles, une de type N' et une autre de type N''. Avec N un nombre entre 1 et 16. Une cellule peut se reproduire en 0, 1 ou 2 cellules filles. Les informations suivantes sont également contenues dans chaque règle de reproduction. (voir figure 4.28):
   - Les modifications qui doivent être effectuées sur chacun des paramètres (paramètres développés au point 2). S'il y a deux cellules filles les modifications de paramètres peuvent être différentes pour chacune des cellules filles.
   - La localisation des deux cellules filles. Ces localisations sont relatives à la position de la cellule mère. Plus concrètement, pour chaque cellule fille, la position est définie parmi une des 8 positions possibles autour de la cellule mère.

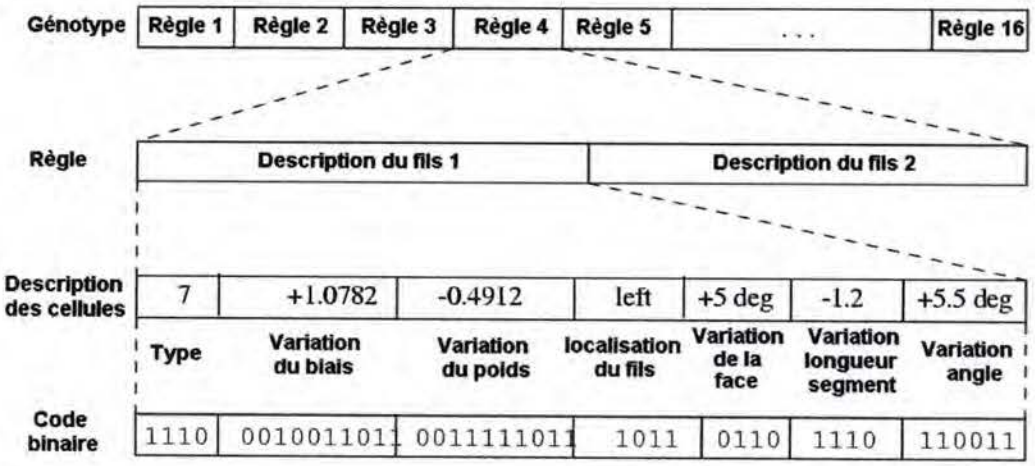


Figure 4-28 : Information contenue dans les règles de divisons

L'espace de croissance des cellules est divisé en trois zones horizontales. Le processus commence par le positionnement de la cellule initiale au centre de cet espace. A la fin des cycles de divisions et de migrations, on obtient au maximum 32 cellules. Ces cellules se spécialisent en neurones, leurs fonctions dépendent de leurs localisations dans l'une des trois zones et du type de la cellule. Un neurone qui se situe dans la bande de bas sera une des entrées du réseau. De la même manière un neurone se trouvant dans la bande du haut sera une unité de sortie et un neurone dans la zone centrale sera un neurone de la couche cachée.

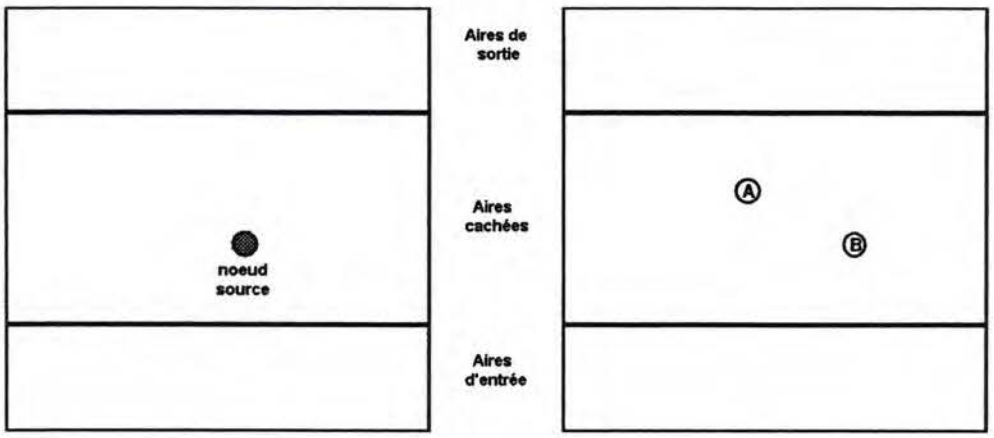


Figure 4-29 : Cellule initiale (a) Première division (b)

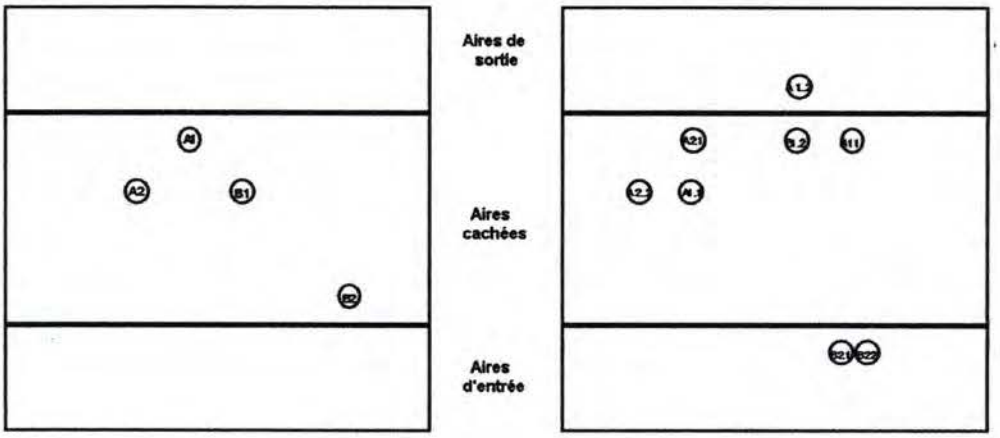


Figure 4-30 : Seconde division (c) et Troisième division (d)

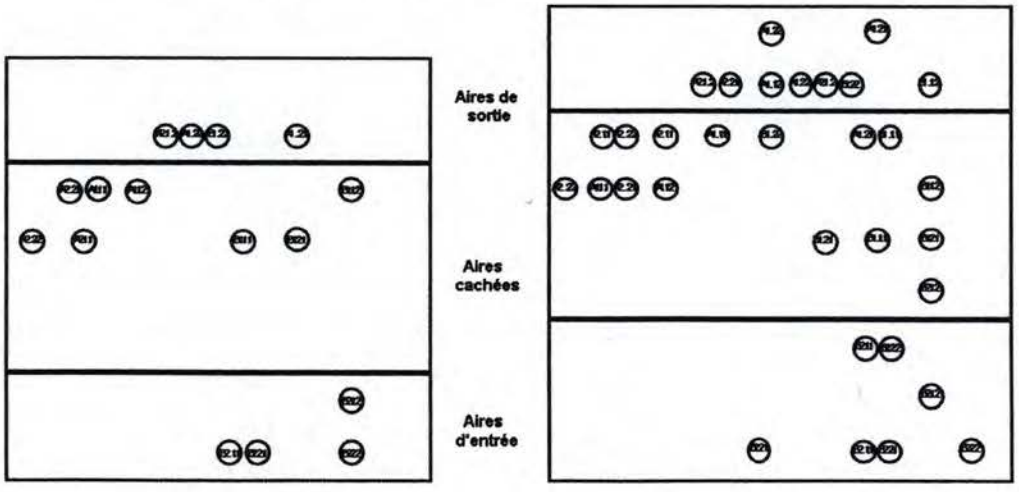
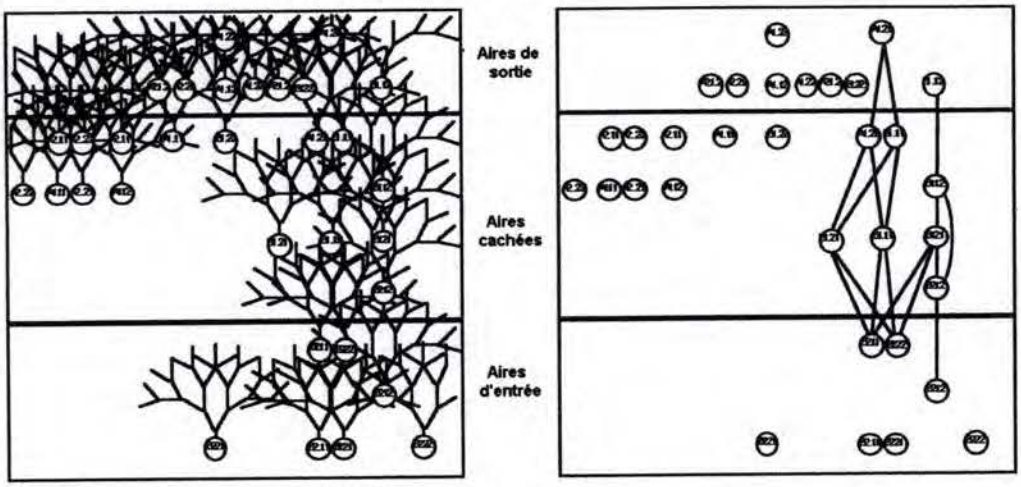


Figure 4-31 : Quatrième division (e) et Cinquième division (f)

A la fin du cycle de divisions et de migrations, le processus de croissance d'axone peut commencer (figure 4.32 (g)). Durant cinq cycles de croissance, chaque neurone développe ses axones en fonction des valeurs correspondantes de leurs angles de branchements et de leurs longueurs de branchement.

Après le processus de croissance si un axone touche une autre cellule, nous obtenons une connexion entre ces deux neurones (figure 4.32 (h)). Des sous réseaux sont ainsi formés. Si ces sous réseaux ne touchent pas une unité d'entrée, ils sont inactifs.



g h

Figure 4-32 : Croissance des axones (g) et Liens entre les neurones (h)

4.3.4.5.2 Algorithme génétique

Une population de cellules initiales est générée, les génomes de chacune de ces cellules souches sont construits aléatoirement.

Après développement, chaque individu de la population est évalué, on regarde si le réseau est capable d'accomplir la tâche cible. Ensuite le processus de reproduction est effectué, les meilleurs individus sont sélectionnés à l'aide des opérateurs des sélections habituelles, croisement et une mutation aléatoire des chaînes de bits.

4.3.4.5.3 Application

Ces auteurs ont utilisé ce modèle pour simuler l'apprentissage d'animaux artificiels vivants dans un environnement contenant deux zones différentes, placées aléatoirement, une zone contenant la nourriture et une zone contenant l'eau. La tâche de chacun des animaux artificiels est d'atteindre la zone de nourriture et d'y rester, aussi longtemps qu'ils ont faim et d'atteindre la zone d'eau et d'y rester aussi longtemps qu'ils ont soif. Chaque animal artificiel est équipé de systèmes de senseurs (entrées du réseau) qui lui permettent de connaître la direction et la distance le séparant du centre de la zone de nourriture et la direction et la distance le séparant du centre de la zone d'eau. Les unités de motivation (entrée du réseau) détectent si un

animal est en état de faim ou de soif. Le signal de sortie indique ce que doit faire l'animal artificiel pour atteindre les zones désirées (11: avancer d'une case, 10: tourner à gauche de 90°; 01: tourner à droite de 90° et 00: rester sur place). Le réseau a donc 5 entrées et deux sorties. Les chercheurs ont obtenu d'excellents résultats avec ce modèle.

4.3.5 Encodage par programmation génétique

4.3.5.1 Génération de réseau de neurones Feedforward - Méthode de Koza

Koza a présenté un modèle intéressant modélisant les réseaux de neurones *feedforward* au moyen d'arbres binaires. Ces arbres binaires sont encodés sur base d'un nouveau type de paradigme, la programmation génétique [Koz91].

4.3.5.1.1 Encodage

Dans ce paradigme, 6 fonctions de base sont utilisées: {P, W, +, -, *, %}.

- La fonction **P** représente un neurone avec une fonction à seuil. Cette fonction calcule un signal de sortie sur base de signaux d'entrée.
- La fonction **W** modélise les poids des connexions entre les neurones.
- Les fonctions arithmétiques **+, -, * et %** sont utilisées pour modifier les valeurs des poids des connexions.

La figure 4.32 montre un exemple de programmes génétiques.

```
(P (W (+ 1.1 0.741) (P (W 1.66 D0) (W -1.387)))
   (W (* 1.2 1.584) (P (W 1.191 D1) (W -0.989 D0))))
```

Figure 4-33: S-Expression LISP

La figure 4.33 représente l'arbre binaire obtenu par le programme génétique de la figure 4.32.

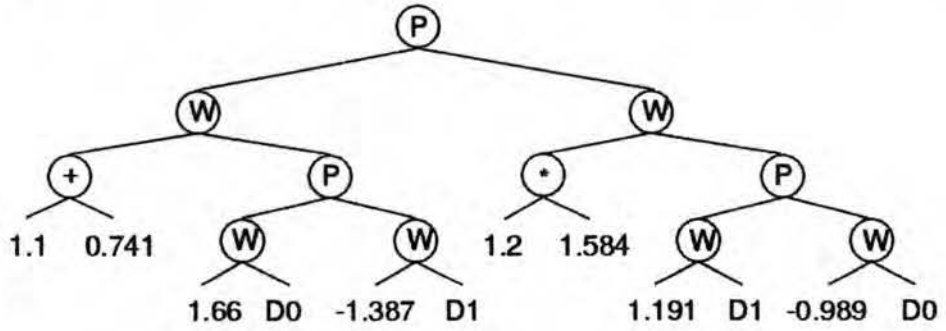


Figure 4-34 : Représentation de l'arbre

La figure 4.34 représente le réseau de neurones résultant de l'arbre binaire de la figure 4.33.

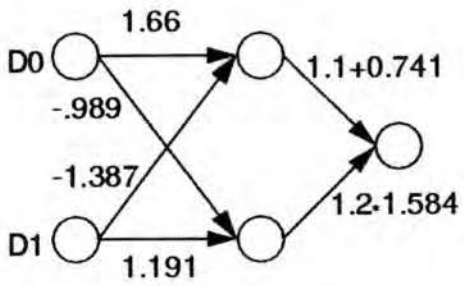


Figure 4-35 : Le réseau de neurones artificiel

Le passage de l'arbre binaire au réseau de neurones s'effectue à l'aide de règles syntaxiques.

Les valeurs terminales des arbres (feuilles) sont soit des constantes réelles, soit des nœuds d'entrées. Les nœuds d'entrées sont représentés par le symbole D_n, par exemple si un réseau de neurones a deux entrées, elles sont notées D_0 et D_1.

Les 6 règles syntaxiques suivantes doivent être respectées.

- Les neurones, nœuds P, ont toujours comme racines de leurs sous-arbres des nœuds W (connexions).
- Les connexions : nœuds W, ont toujours comme racines de leurs sous-arbres gauche, soit un réel, soit un nœud expression {+,-,*,%}. Cette racine de gauche est la valeur du poids de la connexion.
- Les nœuds W ont toujours comme racines de leurs sous-arbres de droite, soit un nœud P, soit un nœud entrée D_n. Ce nœud est le neurone de départ de la connexion.
- Les nœuds réels n'ont pas de sous-arbres.
- Les nœuds expressions {+,-,*,%}ont comme racines de leurs sous-arbres, soit un réel, soit un sous arbre expression
- Aucun arbre ne peut commencer par un nœud W.

Un nœud poids W possède donc deux sous-arbres, celui de gauche qui contient la valeur du poids et celui de droite qui contient le neurone source de la connexion. Le nœud parent de W étant le nœud destination de la connexion.

Il faut remarquer que les signaux d'entrée peuvent apparaître à plusieurs endroits différents dans l'arbre. Il est donc possible de créer une connexion entre un signal d'entrée et plusieurs autres neurones.

Les arbres que nous venons de décrire peuvent représenter des réseaux de neurones avec une seule sortie, qui est la racine de l'arbre. S'il y *n* nœuds de sortie, une racine spéciale, appelée LIST, est utilisée. Cette racine LIST comporte *n* sous-arbres, qui ont chacun comme racine un nœud P, représentant les neurones de sorties. La figure 4.36 montre un exemple de réseau de neurones à deux sorties.

```
(LIST (P (W D1 -1.423) (W D0 (+ 1.2 0.4))
      (P (W D0 (* -1.7 -0.9)) (W D0 (- 1.1 0.5)))).
```

Figure 4-36 : Représentation de S-Expression LISP avec plusieurs nœuds de sortie

Cette fonction LIST vient s'ajouter aux 6 autres fonctions de base définies ci-dessus. Ce nouvel ensemble {P, W, +, -, *, %, LIST} permet de représenter tous les différents types de réseau de neurones feedforward.

4.3.5.1.2 Algorithme génétique

Le croisement s'effectue en échangeant des sous-arbres. Lors de cette opération de croisement, il faut faire attention aux restrictions syntaxiques exposées ci-dessus concernant la succession des nœuds des sous-arbres. Le croisement utilisé est le *croisement avec persistance de structure* (Structure-preserving-crossover). On procède comme suit : Pour le premier parent un nœud est choisi aléatoirement sans aucune restriction. Pour le second parent, un nœud du même type est choisi aléatoirement. Le croisement entre ces deux nœuds est ensuite effectué.

La mutation aléatoire est également utilisée permettant la transformation d'un nœud en un nouveau nœud du même type.

Les types considérés comme identiques sont :

- Les nœuds contenant une fonction P,
- Les nœuds contenant une fonction W
- Les nœuds contenant un signal d'entrée (telle que D_0, D_1)
- Les nœuds contenant une fonction arithmétique ou une constante réelle.

4.3.5.2 Encodage de cellules - Méthode de Gruau

4.3.5.2.1 Introduction

La méthode de Kitano, bien que plus performante que l'encodage direct pour des modélisations d'applications complexes, présente deux gros inconvénients :

- Une matrice de taille *m x m* doit être utilisée pour modéliser un réseau de taille n, avec m la plus petite puissance de 2 plus grande que n.
- Une modélisation d'un réseau acyclique n'utilise pas le triangle inférieur gauche de la matrice *m x m*, mais celui-ci doit cependant être codé.

4.3.5.2.2 Encodage

Pour faire face à ces deux inconvénients, Gruau propose dans son travail [Gru92] un nouveau type de grammaire. Cette grammaire utilise l'encodage en cellules et permet l'encodage de familles de réseaux de neurones de structure similaire. Montrons par

exemple, comment le problème du XOR est modélisé. Pour cette application la fonction *threshold* de 0 ou 1 ainsi que des connexions de poids de -1 ou +1 sont utilisées (Si la somme des poids des entrées est supérieure au *threshold*, la sortie du neurone vaut 1, sinon elle vaut 0). Les entrées de ce réseau peuvent valoir 0 ou 1.

Le passage du génotype au phénotype commence avec une seule cellule appelée cellule ancêtre. Cette cellule possède un pointeur entrée et un pointeur de sortie. Considérons la situation initiale suivante, avec à gauche le graphe des cellules encodées et à droite le réseau de neurones initial (étape 0). A l'étape initiale le pointeur de lecture de la cellule ancêtre est positionné sur la racine de l'arbre. Les registres sont initialisés avec les valeurs par défaut (par exemple le *threshold* est initialisé à 0). Cette cellule va se diviser en d'autres cellules. Une cellule devient un neurone lorsqu'elle perd son pointeur de lecture. Voici, ci-dessous, l'explication des principaux symboles programme de base des cellules encodées. Ce décodage permet de passer du génotype au phénotype.

- Un symbole programme division: créé 2 cellules au départ d'une seule. Dans une divison séquentielle (notée **SEQ**), une division de la cellule parent en deux cellules est effectuée. La première cellule enfant hérite des liens d'entrée de la cellule parent, le second enfant hérite des liens de sortie de la cellule parent. Le premier enfant est connecté au second enfant par une connexion de poids 1. Le lien est orienté du premier enfant vers le second. Comme il y a deux cellules enfants, le symbole de division programme doit avoir 2 cellules descendantes. Le premier enfant déplace son pointeur de lecture vers la branche de gauche de l'arbre, le second enfant déplace son pointeur de lecture vers la cellule de l'arbre de droite. C'est illustré aux étapes 1 et 3 de la figure 4.37.
- Division parallèle (notée **PAR**) est une deuxième sorte de symbole de programme de division. Les deux cellules héritent des entrées et de sorties de la cellule parents (voir étapes 2 et 6). Finalement, quand la cellule se divise, les valeurs des registres de la cellule parent sont recopiées dans les cellules des enfants.
- Le symbole de fin de programme (noté **END**) permet à une cellule de perdre son pointeur et donc de devenir un neurone final, ce qui termine le sous-arbre.
- Modification des registres internes d'une cellule.
  - Le symbole d'incrément **INCBIAIS** (ou de décrément **DECBIAIS**) incrémente de 1 le *threshold* de la cellule
  - Le symbole **INCLR** agit sur le pointeur de sélection. Ce pointeur de sélection pointe sur une des valeurs de la liste des connexions entrant dans le neurone. En exécutant cette instruction, ce pointeur pointe vers la connexion suivante dans la liste. Le symbole **DECLR** a l'effet inverse.

- Le symbole de programme **VAL+** (ou **VAL-**) augmente de 1 (ou diminue de 1) la valeur du poids, dans la liste de poids des entrées du neurone, pointé par le pointeur de sélection.
- Le symbole de programme **CUT** coupe la connexion pointée par le pointeur de sélection. Cet opérateur modifie la topologie du réseau en retirant 1 lien

Il faut remarquer que dans l'exemple ci-dessus le XOR n'illustre pas les opérateurs INCLR, DECLR et CUT.

Dans certains cas, la configuration finale du réseau dépend de l'ordre dans lequel on exécute les instructions, par exemple effectuer l'étape 7 avant l'étape 6 donne un réseau avec 2 poids négatifs au lieu d'un seul, comme dessiné. Le symbole de programme WAIT permet d'attendre 1 tour et donc d'orienter le développement de l'architecture du réseau dans une direction plutôt que dans une autre.

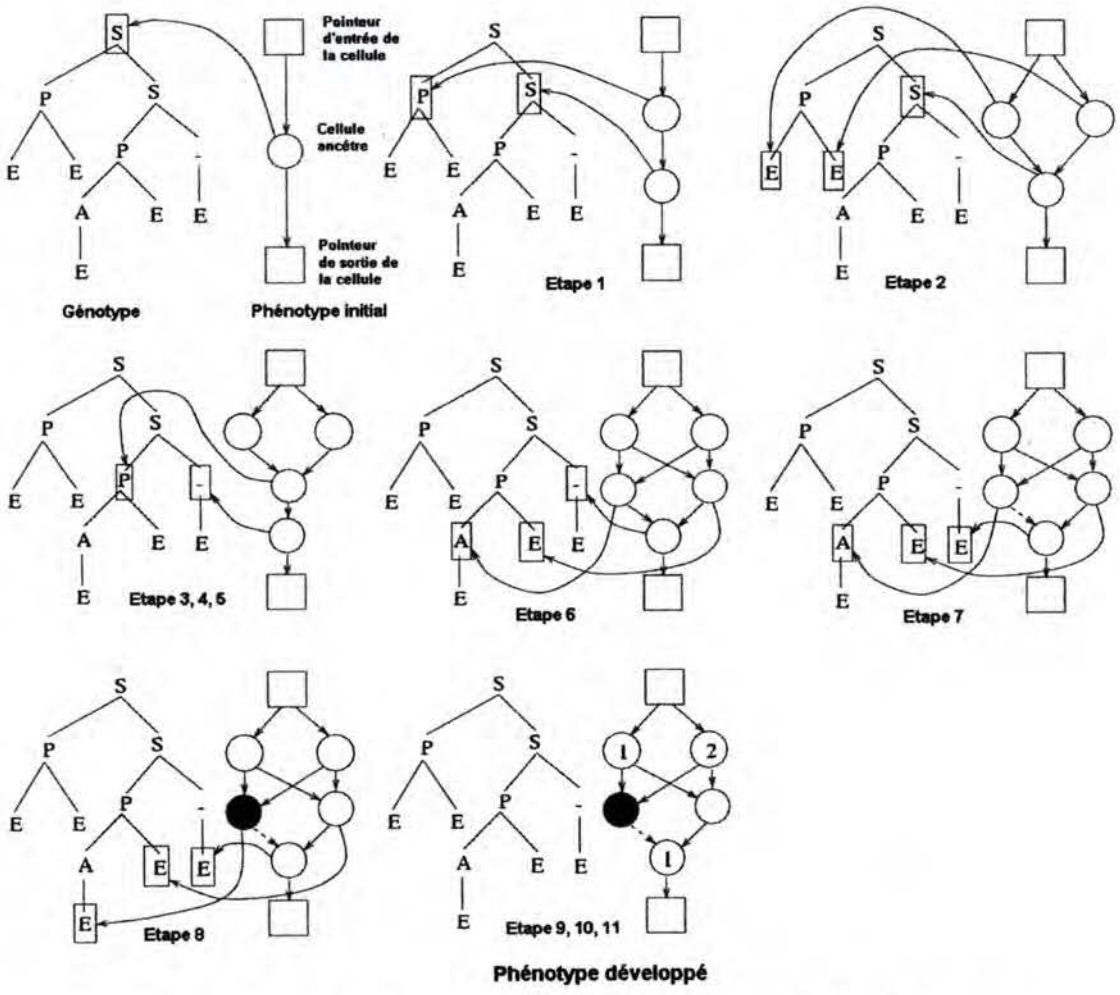


Figure 4-37: Développement du réseau par la méthode de Gruau

4.3.5.2.3 Algorithme génétique

Gruau utilise les opérateurs génétiques de la même manière que Koza dans son ouvrage où il a développé un paradigme pour la programmation génétique [Koz90] et [Koz92]. Les instructions, c'est-à-dire les nœuds de l'arbre, sont mutées en d'autres instructions possédant la même arité. L'opérateur de croisement remplace un des sous arbres d'un parent tiré aléatoirement, par un sous arbre d'un autre parent.

4.3.5.2.4 Le codage de la récursivité

Jusqu'ici la grammaire n'utilise pas de récursivité. Il est important de remarquer, que ce n'est pas parce que la grammaire est récurrente que le réseau de neurones résultant le sera. Le but est ici de développer un réseau qui contiendra la même sous-structure de réseau. Le symbole de programme récurrent *R* est donc introduit. Il contient un nombre de boucle L. La cellule réexécute l'algorithme suivant.

Life = Life-1

If(Life>0) reading-head:=root of the tree

Else reading-head = subtree of the current node

Où life est le registre de la cellule initialisée avec L dans la cellule ancêtre.

Le développement du réseau n'est donc pas stoppé lorsqu'il atteint une valeur limite supérieure déterminée, mais c'est lorsque le code est passé exactement L fois dans l'arbre.

Le nombre L paramétrise la structure du réseau.

4.4 Recherche de règles d'apprentissage de réseau de neurones par algorithmes génétiques

L'idée principale de l'utilisation d'algorithme génétique pour modéliser l'évolution de l'apprentissage est simple : utiliser un génome pour encoder les propriétés dynamiques du réseau plutôt que les propriétés statiques du réseau.

4.4.1 Optimisation de règles d'apprentissage: méthode de Chalmers

David Chalmer a proposé d'utiliser les algorithmes génétiques pour trouver de meilleures règles d'apprentissage de réseaux de neurones [Cha90].

4.4.1.1 Types d'apprentissage

Les trois catégories standards d'apprentissage sont de type supervisé, renforcé et non supervisé. Bien qu'il soit potentiellement possible d'utiliser ces trois catégories d'apprentissage pour effectuer l'évolution de règles d'apprentissage, Chalmers a choisi d'utiliser l'apprentissage supervisé. Il a effectué ce choix car ce type d'apprentissage est le plus facile et le mieux compris des trois.

4.4.1.2 Architecture du réseau

En ce qui concerne la topologie du réseau, les différents poids des connexions du réseau n'ont pas besoin d'être fixés à l'avance, cependant l'architecture (le nombre de couches, le nombre de neurones par couche ainsi que les différentes connexions entre les neurones) doit être fixée et restera inchangée tout au long du processus d'évolution.

Le type de réseau que Chalmers a proposé d'étudier est le réseau à une couche feed-forward. Ce type de réseau présente l'avantage de posséder un algorithme d'apprentissage puissant, la règle des deltas (règle de Widrow).

4.4.1.3 Le codage génétique du mécanisme d'apprentissage

Pour une connexion donnée entre un neurone d'entrée i et un neurone de sortie j, les informations locales incluent 4 éléments.

a_j la valeur de l'activation du neurone d'entrée j

o_i la valeur de l'activation du neurone de sortie i

t_i la valeur cible du neurone de sortie i

w_{ij} la valeur courante du poids de la connexion allant du neurone j au neurone i

Le génome doit encoder une fonction F du type

$$\Delta w_{ij} = F\left(a_j, o_i, t_i, w_{ij}\right)$$

Les hypothèses suivantes ont été prises pour cette fonction F :

- Les 4 variables d'entrée sont indépendantes
- Cette fonction est une fonction linéaire de ces 4 variables
- Cette fonction est également une fonction linéaire du produit de chacune des paires de variables.

F est donc déterminé en spécifiant 10 coefficients. Un 11ème coefficient est cependant utilisé pour la mise à l'échelle des valeurs de cette fonction. On peut remarquer que cette hypothèse n'exclut pas la règle des deltas, qui en est un cas particulier.

$$\Delta w_{ij} = k_0\left(k_1 w_{ij} + k_2 a_j + k_3 o_i + k_4 t_i + k_5 w_{ij} a_j + k_6 w_{ij} o_i + k_7 w_{ij} t_i + k_8 a_j o_i + k_9 a_j t_i + k_{10} o_i t_i\right)$$

Le génome est constitué de 35 bits.

Les 5 premiers bits codent le paramètre de mise à l'échelle k_0, qui prend les valeurs 0, ±1/256, ±1/128, ... , ±32, ±64, via l'encodage exponentiel. Le premier bit encode le signe de k_0 (0=négative et 1=positif), et les 4 bits suivants encodent la valeur. Ces 4 bits sont interprétés comme un entier j entre 0 et 15, nous avons :

$$|k_0| = \begin{cases} 0 & \text{si } j = 0 \\ 2^{j-9} & \text{si } j = 1, \ldots, 15 \end{cases}$$

Les 30 bits suivants encodent les 10 autres paramètres par groupe de 3 bits. Le premier bit de chacun des groupes de trois bits représente le signe et les deux autres représentent une valeur parmi 0 [00], 1 [01], 2 [10] ou 3 [11] à l'aide du même type d'encodage exponentiel que pour k_0.

$$|k_i| = \begin{cases} 0 & \text{si } j = 0 \\ 2^{j-1} & \text{si } j = 1, 2, 3 \end{cases}$$

Par exemple la règle des deltas peut être exprimée génétiquement comme

11011 000 000 000 000 000 000 000 010 110 000

Les coefficients décodés sont :

4 0 0 0 0 0 0 0 -2 2 0

Et la formule résultante est :

$$\begin{aligned} \Delta w_{ij} &= 4(-2a_j o_i + 2a_j t_i) \\ &= 8\, a_j\, (t_i - o_i) \end{aligned}$$

4.4.1.4 Exemple d'un apprentissage

Apprentissage d'un réseau FeedForward avec une couche d'entrée de 5 neurones (a_i), une couche de sortie de 8 neurones, et pas de couches cachées. Ce réseau peut être considéré comme 8 réseaux distincts avec 5 entrées et une sortie étant donné que nous n'avons pas de couches cachées.

Pour chaque tâche (neurone de sortie t_i) nous avons 12 exemplaires

a_1	a_2	a_3	a_4	a_5	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
1	1	1	1	1	1	0	0	0	1	0	1	0
0	0	0	0	0	0	0	1	0	0	1	0	1
0	1	1	1	0	0	0	1	0	1	0	1	0
1	1	0	0	0	0	1	1	1	0	1	1	1
1	0	1	0	1	1	0	0	0	0	0	1	1
0	1	1	0	0	0	0	1	0	1	1	0	1
0	1	1	1	1	1	0	1	0	1	0	1	0
0	1	0	0	0	0	0	1	1	1	1	0	1
1	1	0	0	1	1	0	0	1	0	1	1	1
1	0	0	1	0	0	0	1	1	1	0	1	1
1	0	1	1	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	0	0	1

4.4.1.5 Evaluation du Fitness

Pour évaluer la fitness la procédure suivante est appliquée. Des tâches sont sélectionnées aléatoirement (typiquement 20) parmi les 96 tâches d'apprentissage (c-à-d 8 x 12). Les fitness sont obtenues comme suit :

(1) Création d'un réseau avec un nombre d'entrées approprié pour la tâche et un neurone de sortie.

(2) Initialisation des poids des connexions du réseau aléatoirement entre -1 et 1

(3) Pour un nombre d'époques déterminé (typiquement 10), tester successivement les exemplaires d'apprentissage, et pour chacun à la suite l'un de l'autre faire :

(3a) Propager les signaux d'entrée jusqu'aux sorties et comparer la sortie obtenue avec la sortie désirée

(3b) Ajuster les poids du système selon la formule spécifiée par la procédure d'apprentissage, sur base des entrées, des sorties, des signaux d'apprentissage et des poids actuels.

(4) A la fin de ce processus la fitness est obtenue en testant le réseau avec l'ensemble des exemplaires d'apprentissage, n divisant la somme des erreurs par le nombre d'exemplaires, en soustrayant à 1 cette valeur et en multipliant par 100 on obtient le pourcentage de fitness.

Il est important de remarquer que vu le petit nombre d'itérations c-à-d 10, le fait d'avoir une fitness de 98% est excellent. Pour pouvoir atteindre 100% ce nombre d'itérations doit être plus élevé.

4.4.1.6 Paramètres de l'algorithme génétique

La taille de la population est de 40 individus chacun constitué de 35 bits.

La reproduction est de type: proportionnel au Fitness.

Le type de croisement est un croisement à 2 points avec un taux de croisement de 0.8

La mutation de bits est également utilisée avec un taux de 0.01

L'élitisme est utilisé, conservant ainsi pour chaque génération le meilleur individu.

Le nombre de générations est de 1000

4.4.1.7 Résultats obtenus

Au début de l'algorithme les Fitness sont entre 40% et 60% indiquant que l'apprentissage n'est pas significatif. Après 1000 générations, le fitness maximum est typiquement entre 80% et 98% avec une moyenne de 92%. Le tableau ci-dessous donne les résultats pour 10 essais consécutifs avec des poids de départ différents

Tests	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	Fitness
1	0.25	0	0	0	0	0	0	0	-4	4	0	89.6%
2	-2.00	0	0	0	0	0	0	0	2	-2	0	98.0%
3	0.25	0	-1	-2	4	0	0	0	-2	4	-2	94.3%
4	0.25	0	-1	-2	4	0	0	0	-2	4	-2	92.9%
5	-0.25	0	0	1	-1	0	1	-1	4	-4	0	89.8%
6	-1.00	0	0	-1	1	0	0	0	4	-4	0	97.6%
7	4.00	0	0	1	-1	0	0	0	-2	2	0	98.3%
8	-0.06	0	0	0	-2	-1	2	2	4	-4	2	79.2%
9	-0.25	0	0	2	-1	0	-1	-1	2	-4	0	89.8%
10	0.25	0	-1	-2	4	0	0	0	-2	4	-2	93.2%

La solution obtenue pour le test numéro 2 est une version bien connue de la formule de la règle des deltas (ou règle de Widrow-Hoff). La règle d'apprentissage est :

$$\Delta w_{ij} = -2\,(2\,a_j o_i - 2\,a_j t_i)$$

$$= 4\,a_j\,(t_i - o_i)$$

d'autres résultats comme le test numéro 7

$$\Delta w_{ij} = 4\,(o_i - t_i - 2\,a_j o_i + 2\,a_j t_i)$$

$$= 8\,(a_j - 0.5\,)\,(t_i - o_i)$$

et le test numéro 6

$$\Delta w_{ij} = -1\,(-o_i + t_i + 4\,a_j o_i - 4\,a_j t_i)$$

$$= 4\,(a_j - 0.25\,)\,(t_i - o_i)$$

en sont des variantes

4.4.2 Optimisation de taux d'apprentissage γ et du momentum α pour la rétropropagation du gradient BP

L'algorithme de la rétropropagation du gradient BP utilise deux paramètres fortement liés entre eux, le taux d'apprentissage η et le momentum α. Le choix de ces deux paramètres influence fortement la bonne convergence du problème, mais pour l'instant il relève plus d'un art que réellement d'une science. Ce choix est donc loin d'être trivial. Comme les algorithmes génétiques ont obtenu de grands succès pour l'optimisation de fonctions fortement non-linéaires comme celle-ci, certains chercheurs ont essayé de les utiliser [BMIS90].

Le domaine dans lequel se situe le taux d'apprentissage η et de momentum α sont :

$$0 \leq \gamma \leq 8 \quad \text{et} \quad 0 \leq \alpha \leq 1.0$$

Les différents paramètres suivants pour l'algorithme génétique ont été utilisés pour résoudre le problème bien connu de la symétrie pour 6 bits (Un vecteur de taille 2N est présenté au réseau possédant 2N entrées, ce réseau possède une seule sortie binaire qui vaut vrai si pour tout i = 1 jusque N, l'entrée Ii est égale à l'entrée I_{2N-I+1})

- Chacun des 2 paramètres a été codé sur 10 bits
- La population a une taille de 5
- Le nombre de générations est de 200
- La valeur du fitness est l'opposé (valeur négative de) la somme des erreurs au sens des moindres carrés.

La figure ci-dessous montre les résultats obtenus après 200 générations les cercles sont inversement proportionnels à la somme des erreurs au sens des moindres carrés.

Les valeurs habituellement utilisées pour ces deux paramètres sont $\gamma = 0.1$ et $\alpha = 0.2$ mais l'algorithme génétique converge dans la majorité des cas vers $\gamma = 2.5$ et $\alpha = 0.33$

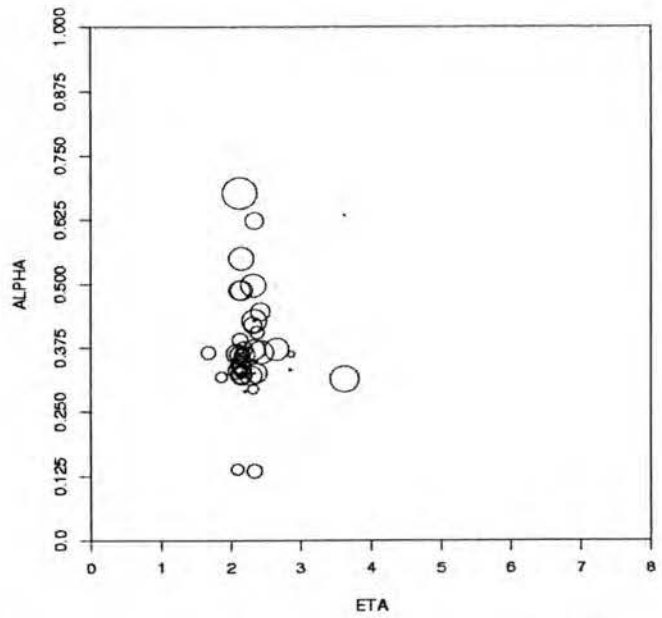


Figure 4-38: η et α trouvés avec l'algorithme génétique

Les valeurs pourraient être assez surprenantes cependant le nombre d'itérations utilisé est excessivement faible, en général pour le problème de la symétrie pour 6 bits les valeurs de 0.1 et 0.2 convergent bien pour des itérations de l'ordre de 4000. L'ordre dans lequel les éléments d'apprentissage sont présentés au réseau (choisi aléatoirement ou présenté toujours dans la même séquence les uns à la suite des autres) a également une grande influence sur les performances de l'algorithme pour ce problème.

4.4.3 Autres applications

D'autres approches, différentes des précédentes ont été proposée comme celle de R. Krishnan et B. Ciesielski [KrCi94]. L'encodage porte sur des règles de variations purement empiriques, plutôt que sur les poids eux-mêmes ou sur une règle de variations de poids plus classique comme celle proposée par Chalmers ci-dessus.

4.5 Algorithmes génétiques pour la sélection des entrées lors d'une phase d'apprentissage de réseau de neurones

4.5.1 Sélections des neurones d'entrées d'un réseau de neurones

Pour beaucoup d'applications de réseau de neurones le nombre d'entrées peut être très important. Parfois certaines de ces entrées sont inutiles voir redondantes. Un grand nombre d'entrées augmente la taille du réseau, augmente la durée d'apprentissage ainsi que l'effort nécessaire pour se procurer ces différentes entrées. Une réduction de ce nombre d'entrées est parfois très utile et bénéfique.

Le problème à résoudre ici est de trouver un set de solutions en diminuant le nombre d'entrées sans pour autant diminuer les performances du réseau pour l'application désirée.

Les algorithmes génétiques ont été utilisés pour effectuer cette tâche et de très bons résultats ont été obtenus. Dans le processus d'évolution sur l'ensemble des différentes entrées, un vecteur binaire de longueur égale au nombre d'entrées est utilisé. Dans ce vecteur la valeur "1" indique la présence de l'entrée dans le réseau, la valeur "0" indique l'absence de cette entrée dans le réseau.

Le processus d'évolution est obtenu en effectuant l'apprentissage du réseau sur base de ces différents vecteurs d'entrées. En général le réseau utilisé pour cet apprentissage a une structure fixe.

En général pour ce type de recherche l'espace est très grand par exemple pour un réseau de 64 entrées, il y a 2^{64} solutions possibles. Une recherche par algorithme génétique est très propice pour ce type de problème.

Afin d'effectuer l'apprentissage de ces réseaux la rétro-propagation du gradient BP est souvent utilisée pour savoir à quel moment les itérations doivent être arrêtées, la *Cross-Validation* est parfois utilisée. La *Cross-Validation* teste durant le processus d'itérations la valeur de l'erreur sur un groupe de tests de validation (voir figure 4.39)

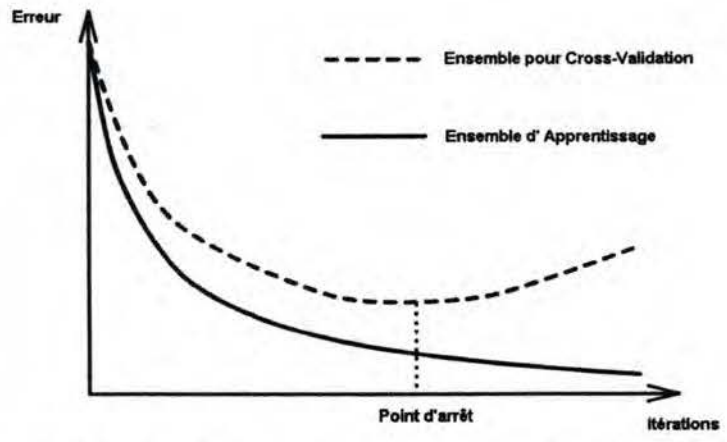


Figure 4-39: Courbes d'apprentissage et courbes de Cross-Validation

Lorsque cette valeur augmente significativement, le processus d'itération doit stopper car si l'on continue, le réseau de neurones obtenu verra ses propriétés de modélisation se détériorer. En général le groupe d'éléments de validation est différent du groupe d'éléments de test.

Une application très intéressante, dans le cadre de la reconnaissance vocale, où le nombre d'entrées était assez important (153) a pu être réduit à 33 (facteur 5) en utilisant cette méthode [ChLp91]. Dans leurs travaux Chang et Lippmann ont effectué la classification en reconnaissance vocale pour la syllabe "e" en anglais. Il y a 9 types différents de mots utilisant la prononciation "e" en anglais. Les signaux d'entrées du réseau sont obtenus en effectuant une analyse spectrale du signal sonore en fonction de différents critères. Ces chercheurs ont utilisé 10 éléments pour l'apprentissage et 16 éléments pour la *cross-validation*. Les résultats qu'ils ont obtenus après 12 000 itérations sont les suivants, pour l'apprentissage l'erreur est de 3.3% et pour le test l'erreur est de 17.5% (voir figure 4.40).

D'autres succès ont également été obtenus dans le domaine de la détection de problèmes dans les centrales nucléaires [WeSuTh95] et [GuUh92].

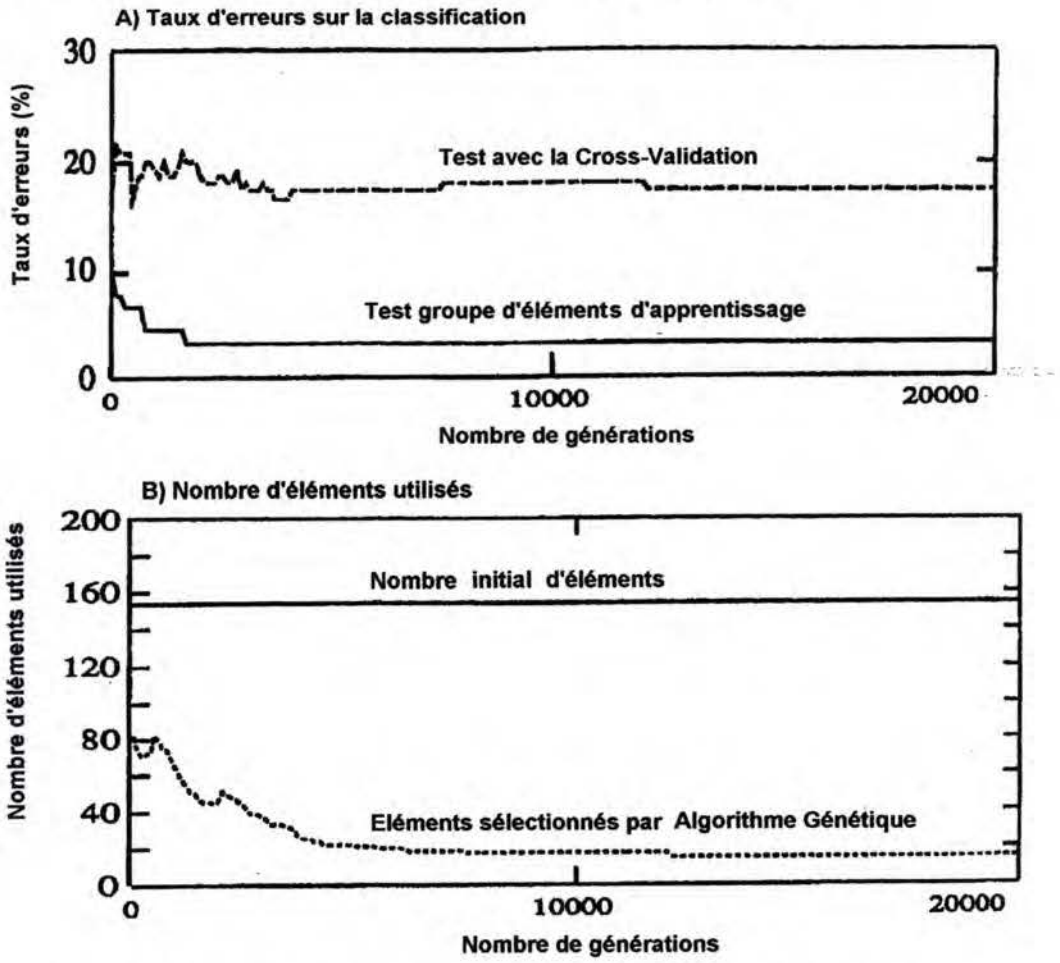


Figure 4-40: Progression d'un algorithme génétique pour la classification de la voyelle "E" au départ d'un réseau de neurones de 153 entrées (partie 2)

4.6 Sélections des éléments du groupe d'apprentissage de réseau de neurones

Les performances d'un réseau de neurones utilisé pour effectuer de la classification d'éléments, augmentent en général, plus la taille du groupe de ces éléments d'apprentissage augmente. Mais souvent la durée d'apprentissage ainsi que la taille de la mémoire nécessaire pour traiter ces éléments augmentent également. Ces deux points sont donc antagonistes.

Une solution pour résoudre ce problème est de diminuer la taille de ce groupe d'apprentissage en ne conservant que les éléments pertinents.

Les algorithmes génétiques vont donc être utilisés pour effectuer cette tâche. A la chaîne de bits utilisée pour effectuer le codage, un facteur k est ajouté, composé de 3 bits. Ce facteur est inversement proportionnel au nombre d'éléments du groupe d'apprentissage utilisé.

Dans leurs travaux Chang et Lippmann [ChLp91] ont proposé la résolution de problèmes pour la classification en 10 grandes catégories de mots, commençant par "h" et finissant par "d", avec une voyelle sonore entre ("head", "hid", "hod", "had", "hawed", "heard", "heed", "hud", "who'd" et "hood"). Un total de 338 éléments constitue le groupe d'apprentissage et 333 éléments constitués le groupe de test. Chaque individu est constitué de 2 éléments qui sont deux fréquences caractérisant la voyelle obtenue par analyse spectrographique.

L'erreur après apprentissage sur le groupe d'apprentissage est très faible cependant l'erreur sur le groupe de test est plus élevée: 25% avec les 333 éléments.

D'autres méthodes existent pour condenser en un même point des entrées voisines mais ne permettent pas de réduire de beaucoup la mémoire de stockage et en plus ne permettent de réduire le groupe d'éléments d'apprentissage que d'un facteur 2 (de 338 à 152). Pour plus de détails le lecteur pourra se référer à [NgLi91].

Après apprentissage par algorithme génétique le groupe d'éléments d'apprentissage a été réduit par un facteur 8 (de 338 à 43) avec une erreur sur le groupe de test de 20.1%. L'apprentissage obtenu avec ce plus petit groupe d'éléments d'apprentissage est donc meilleur qu'avec le groupe d'origine.

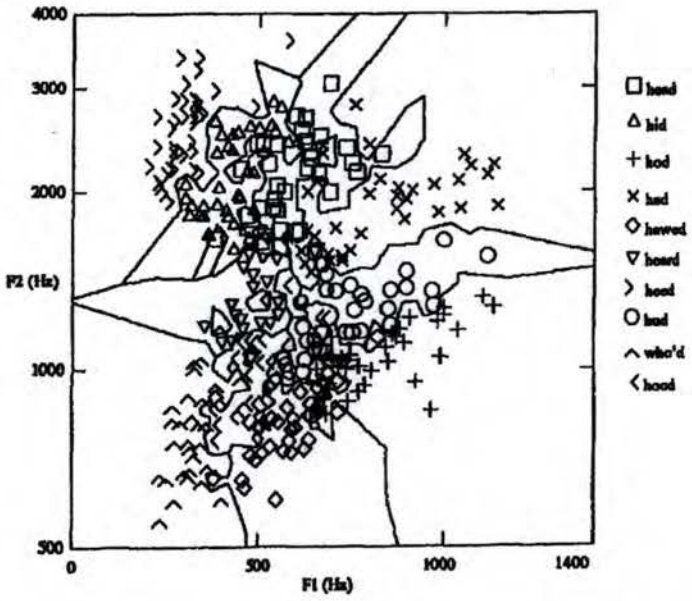


Figure 4-41: Frontière pour la classification pour le problème de la voyelle sonore avec un groupe d'apprentissage de 338 éléments

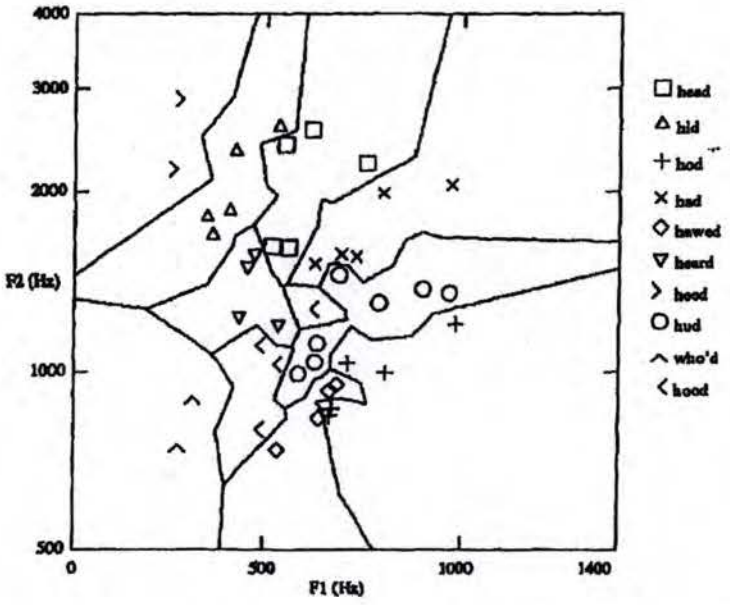


Figure 4-42: Frontière pour la classification pour le problème de la voyelle sonore avec sélection du groupe d'apprentissage de 43 éléments par GA

5 Chapitre 4 : Programmes

Le programme pour résoudre les problèmes d'apprentissage de réseau de neurones NN à l'aide d'algorithme génétique GA a été réalisé en Java. Il est constitué d'une série de class réparties en 5 packages différents. Ces packages sont:

- Package GA: Ce package regroupe toutes les class gérant l'algorithme génétique.
- Package NN: Ce package contient les différentes class de construction, d'apprentissage et d'utilisation de réseaux de neurones.
- Package GA_NN: Ce package implémente les différentes class intégrant les algorithmes génétiques aux réseaux de neurones.
- Package Tools: Ce package regroupe les class utilitaires communes aux différentes autres class.
- Package Test: Ce package contient les class de test des différentes applications.

5.1 Package GA

5.1.1 Package GA.Individu

5.1.1.1 Indiv_Abstract.class

La première class de ce package est une interface qui définit l'ensemble des méthodes que doivent contenir toutes les class concrètes qui implementent cette interface ou une de ses class dérivées.

5.1.1.2 Individu.class

Cette class abstraite est dérivée de l'interface `Indv_Abstract.class`. Elle contient les différentes variables et méthodes propres aux différents individus: Fitness, nombre de chromosomes, ...

5.1.1.3 Indiv1.class

Cette class concrète est dérivée de la class abstraite `Indivudu.class`. Ses chromosomes sont représentés par une suite de réels. Ces réels appartiennent à un domaine borné supérieurement et inférieurement. La méthode la plus importante de cette class est `Indiv_Eval()`, qui permet d'évaluer la fitness de chaque individu sur base des variables encodées dans les chromosomes

5.1.1.4 Indiv_Bin1.class

Cette class concrète est dérivée de la class abstraite `Indivudu.class`. Ses chromosomes sont représentés par une chaîne de bits. La méthode la plus importante est également `Indiv_Eval()` comme pour la class `Indiv1.class`.

5.1.1.5 Indiv1_BP_Param.class

Cette class concrète est dérivée de la class concrète `Indiv1.class`. Elle est constituée de 2 chromosomes contenant chacun un réel. Le premier chromosome est le *taux de croissance*. Il peut prendre des valeurs comprises entre 0 et 20. Le second chromosome est le *momemtum*. Il peut prendre des valeurs comprises entre 0 et 1. Ces deux paramètres sont utilisés dans l'algorithme de la rétropropagation du gradient.

La méthode d'évaluation est redéfinie et permet pour chaque individu de calculer les sorties du réseau au départ des entrées à l'aide de la méthode de rétropropagation du gradient (BP backpropagation du gradient):

Les valeurs des deux chromosomes sont chargées dans le réseau de neurones `NN` à l'aide des méthodes suivantes:

```
NN.BackPropagation.NN_BackPropagation.Set_Epsilon( Chrom[0] )
NN.BackPropagation.NN_BackPropagation.Set_Momentum( Chrom[1] )
```

La rétropropagation du gradient est calculée sur 20 itérations à l'aide des 3 méthodes suivantes

```
NN.FeedForward.NN_FeedForward.UpDateInput(NN, NN_Pattern_In, u)
NN.FeedForward.NN_FeedForward.UpDateLayer(NN,h)
NN.BackPropagation.NN_BackPropagation.UpDateWeightandThresh(NN_Pattern_In, u)
```

Chaque calcul est effectué 50 fois successivement pour avoir une moyenne de l'erreur en fonction de différentes valeurs des poids initiaux choisies aléatoirement entre -1 et 1 (afin de ne pas favoriser un seul bon tirage). Ces poids `Val[]` sont placés dans le réseau `NN` avant chaque calcul à l'aide de la méthode suivante

```
Ga_Nn.Opt_Weight.NN_Weight_Modif( NN, Val)
```

Le réseau `NN` est chargé avec un fichier source `Nomfich` à l'initialisation grâce à la méthode

```
NN.Struct_In.NN_Weight_Struct_In.NN_Weight_Struct_In(String Nomfich).
```

Le groupe d'éléments d'apprentissage `File_Data` est chargé grâce à la méthode

```
NN.Pattern.NN_Pattern_In.NN_Pattern.NN_Pattern_In(String File_Data)
```

L'erreur est calculée à l'aide de la fonction

```
NN.FeedForward.NN_FeedForward.LessSqrError(NN, NN_Pattern_In, u)
```

5.1.1.6 Indiv1_NN.class

Cette class concrète est dérivée de la class concrète `Indiv1.class`, elle est constituée de m chromosomes contenant chacun un réel (m étant égale à la somme du nombre de

poids et du nombre de offsets). Chaque chromosome peut prendre des valeurs entre -20 et 20. Ces paramètres sont utilisés dans l'algorithme de la rétropropagation du gradient BP.

La méthode d'évaluation est redéfinie et permet pour chaque individu d'optimiser les poids et les offsets du réseau à l'aide de la méthode de rétropropagation du gradient.

Les valeurs des chromosomes `Chrom[]` sont chargées dans le réseau de neurones `NN` à l'aide des méthodes suivantes:

```
Ga_Nn.Opt_Weight.NN_Weight_Modif(NN, Chrom)
```

La rétropropagation du gradient est effectuée de la même manière qu'au point précédent. Le réseau `NN` ainsi que le groupe d'apprentissage sont chargés de la même manière qu'au point précédent

5.1.1.7 Indiv1_Bin1_NN.class

Cette class concrète est dérivée de la class concrète `Indiv_Bin1.class`, elle est constituée d'une suite de *m* x *m* bits (m est égale au nombre de neurones du réseau). Ce vecteur encode directement le réseau en fonction des différentes connexions (méthode de Miller). L'apprentissage se fait par rétropropagation du gradient.

La méthode d'évaluation est redéfinie. Elle permet pour chaque individu d'évaluer, si l'architecture du réseau est optimale

Les valeurs des chromosomes `Chrom_Bin[]` sont chargées dans le réseau de neurones `NN` à l'aide des méthodes suivantes:

```
Ga_Nn.Opt_Struct.NN_Struct_Modif(Chrom_Bin, Nbr_Neur)
```

La rétropropagation du gradient est effectuée de la même manière qu'au point précédent. Le réseau `NN` ainsi que le groupe d'apprentissage sont chargés de la même manière qu'au point précédent.

5.1.2 Package GA.Population

5.1.2.1 Pop_Abstract.class

Cette class est une interface qui définit l'ensemble des méthodes que doivent contenir les class concrètes qui implémentent cette interface ou une de ses class dérivées.

5.1.2.2 Population.class

Cette class abstraite est dérivée de l'interface `Pop_Abstract.class` . Elle est constituée d'une suite d'individus. Les méthodes principales de cette class sont :

- **`void`** `ModifNombrePop(`**`int`** `i)` qui modifie le nombre d'individus de la population
- **`void`** `ModifTauxMut(`**`double`** `i)` qui modifie le taux de mutation

- **void** `ModifTauxCrois(double i)` qui modifie le taux de croisement
- **void** `Scaling(String d)` qui active le scaling avec comme paramètre d, qui peut être égal `"Lin"` linéaire, `"Exp"` exponentiel ou `"Position"` de position
- **void** `Simulated_Annealing_activation()` active le recuit simulé
- **void** `Elitisme_Set(double i)` active l'élitisme avec comme paramètre un taux d'élitisme égal à i
- `Population Initialisation()` initialise la population
- **void** `Evaluation()` effectue l'évalation des individus de la population
- **void** `Selection()` sélectionne les individus sur base de l'algorithme de la `Roulette Whell`
- **void** `Croisement()` effectue un croisement de la population
- **void** `Mutation()` effectue une mutation de la population

5.1.2.3 Popu1.class

Cette class concrète implémente la class abstraite `Population.class`. Les méthodes principales sont :

- **protected void** `CrossOver_Individus(Indiv_Abstract Chr_P1, Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1, Indiv_Abstract Chr_F2)` méthode de croisements de réels Floating CrossOver
- **protected void** `Mutation_Individu(Indiv_Abstract Chr_P, Indiv_Abstract Chr_F)` méthode de mutations de réels Floating Mutation
- **public** `Indiv_Abstract new_Individu()` qui spécifie le type d'individu : `Indiv1`

5.1.2.4 Popu_Bin1.class

Cette class concrète implémente la class abstraite `Population.class`. Les méthodes principales sont :

- **public** `Indiv_Abstract new_Individu()` qui spécifie le type d'individu : `Indiv_Bin1`
- **protected void** `CrossOver_Individus(Indiv_Abstract Chr_P1, Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1,Indiv_Abstract Chr_F2)` méthode de croisements à un point ou à deux points de chaînes de bits
- **protected void** `Mutation_Individu(Indiv_Abstract Chr_P, Indiv_Abstract Chr_F)` méthode de mutations de chaînes de bits

5.1.2.5 Popu1_Ga_NN_BP_Param.class

Cette class concrète implémente la class concrète `Popu1.class`. La méthode principale est :

- **public** `Indiv_Abstract new_Individu()` qui spécifie le type d'individu : `Indiv1_BP_Param`

5.1.2.6 Popu1_Ga_NN_Davis.class

Cette class concrète implémente la class concrète `Popu1.class`. Les méthodes principales sont :

- **public** `Indiv_Abstract new_Individu()` qui spécifie le type d'individu : `Indiv1_NN`
- **protected void** `CrossOver_Individus(Indiv_Abstract Chr_P1, Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1,Indiv_Abstract Chr_F2)` méthode de croisements des noeuds

5.1.2.7 Popu_Bin1_Struct2.class

Cette class concrète implémente la class concrète `Popu_Bin1.class`. Les méthodes principales sont :

- **public** `Indiv_Abstract new_Individu()` qui spécifie le type d'individu : `Indiv1_Bin_NN`
- **protected void** `Mutation_Individu(Indiv_Abstract Chr_P, Indiv_Abstract Chr_F)` méthode de mutations de réseaux codés avec la méthode de l'encodage direct

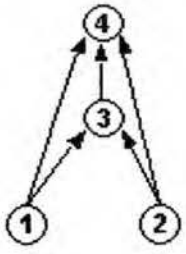
5.2 Package NN

5.2.1 Package NN. Struct_In

5.2.1.1 NN_Struct_In.class

Cette class permet de connaître l'architecture d'un réseau au départ d'un fichier décrivant les différentes couches, les neurones contenus dans ces couches, ainsi que les neurones sources de ces neurones. Ci-dessous, un fichier type décrit le réseau ci-contre

```
"Samples.Xor_v02.Struct_In.txt"
      Nombre total de neurones :
      4
      Couche n°:
      0
      Numero Neurone et Neurones sources :
        1
      Numero Neurone et Neurones sources :
        2
      Couche n°:
      1
      Numero Neurone et Neurones sources :
        3   1 2
      Couche n°:
      2
      Numero Neurone et Neurones sources :
        4   1 2 3
```



5.2.1.2 NN_Weight_Struct_In.class

Cette class permet de construire un réseau au départ d'un fichier contenant les différents vecteurs sources. Ci-dessous, un fichier type, qui décrit également le réseau de la figure 6.1

```
"Samples.Xor_v02.Weight_Struct_In.txt"
      Weight
      0.145 -0.011 0.236 -0.064 -0.171
      Thresh
      -5.0 -5.0 0.546 -0.470
      Nbr_Input
      2
      Nbr_Output
      1
      VC
      1 3 4
      VP
      1 1 1 3 6
      VS
      1 2 1 2 3
      VP_To
      1 3 5 6 6
      VS_To
      3 4 3 4 4
```

5.2.2 Package NN.Struct_Out

Ce package contient la class `NN_Struct_Out.class`, cette class permet de construire un fichier contenant un réseau comme celui du point précédent (avec les différents vecteurs sources).

5.2.3 Package NN.Pattern

Ce package contient la class `NN_Pattern_In.class`, cette class permet de lire un fichier contenant le groupe des éléments d'apprentissage. Ci-dessous, un fichier type contenant un groupe de 5 éléments d'apprentissage et un groupe de 5 éléments de test (dans ce cas ces deux groupes sont identiques).

```
"Samples.Xor_v02.Pattern.txt"
      Nbr de neurones In:
      2
      Nbr de neurones Out:
      1
      Nbr de samples:
      4
      Nbr de Test:
      4
      1.0  1.0  0.0
      0.0  1.0  1.0
      1.0  0.0  1.0
      0.0  0.0  0.0
      0.0  0.0  0.0
      1.0  0.0  1.0
      0.0  1.0  1.0
      1.0  1.0  0.0
```

5.2.4 Package NN.Struct

Ce package contient la class `NN_Struct.class`, cette class contient toutes les méthodes permettant de construction le réseau de neurones. Le réseau est encodé à l'aide des vecteurs des précédents et des suivants (`VP`, `VP_To`, `VS` et `VS_To`). Il contient le vecteur de poids `Weight[]`, le vecteur des offsets `Thresh[]`, le vecteur des états de sorties des neurones `State[]`, le vecteur des activations d'entrées des neurones `Actfn[]`.

5.2.5 Package NN.FeedForward

Ce package contient la class `NN_FeedForward.class`. Les méthodes principales de cette class sont :

- `UpDateInput(NN,NN_Pattern_In, u)` cette méthode permet de calculer les sorties d'un réseau `NN` sur base des entrées d'un individu `u` d'un groupe d'apprentissage `NN_Pattern_In`.
- `UpDateLayer(NN,h)` calcule en série chacune des h couches du réseau `NN`
- `LessSqrError(NN, NN_Pattern_In, u)` calcule l'erreur au sens des moindres carrés du réseau `NN` pour l'élement u du groupe d'apprentissage `NN_Pattern_In`

5.2.6 Package NN.BackPropagation

Ce package contient la class `NN_BackPropagation.class`. Les méthodes principales de cette class sont :

- `UpDateWeightandThresh(NN_Pattern_In, u)` qui effectue une backpropagation sur base des sorties espérées de l'élement u du groupe d'apprentissage `NN_Pattern_In`.
- `Set_Epsilon(d)` remplace le taux d'apprentissage Epsilon par le réel d
- `Set_Momentum(d)` remplace le momentum par le réel d

5.3 Package GA_NN

5.3.1 Package GA_NN.Opt_Struct

5.3.1.1 NN_Initialisation.class

Cette class permet d'initialiser un vecteur `p[]` par encodage direct, à l'aide du nombre de neurones `Nbr_Neur`, du nombre de neurones d'entrée `Nbr_In` et du nombre de neurones de sortie `Nbr_Out`.

```
public NN_Initialisation(int p[], int Nbr_Neur, int Nbr_In, int
Nbr_Out)
```

5.3.1.2 NN_Struct_Modif.class

Cette class permet de créer des connexions du réseau au départ d'un vecteur de connexions `t[]` (à l'aide de la première méthode ci-dessous) et renvoit le réseau ainsi créé, de type `NN_Struct` (à l'aide de la seconde méthode ci-dessous).

```
public NN_Struct_Modif(int t[], int Nbr_Neur)
public NN_Struct Get_NN_New()
```

5.3.1.3 NN_CrossOver.class

Cette class permet d'effectuer un croisement du vecteur `p[]`

```
public NN_CrossOver(int p1[], int p2[], int v, int Nbr_Neur)
```

5.3.1.4 NN_Mutation.class

Cette class permet d'effectuer une mutation du vecteur `p[]`

```
public NN_Mutation(int p[], int Nbr_Neur, int Nbr_In, int Nbr_Out)
```

5.3.2 Package GA_NN.Opt_Weight

5.3.2.1 NN_Weight_Modif.class

Cette class contient deux méthodes principales

`public NN_Weight_Modif(NN_Struct NN, double t[])` Permet d'encoder dans le réseau NN, les poids et le offset stockés dans le vecteur de réel t[]

`public void NN_Weight_Modif_Back(NN_Struct NN, double t[])` Permet d'encoder dans le vecteur de réel t[] les poids et les offsets stockés dans le réseau `NN`

5.3.2.2 NN_CrossOver.class

Cette class contient la méthode principale suivante

public void NN_CrossOver_Node(**int** N, **double** t1[], **double** t2[])
Effectue un CrossOver des $N^{ième}$ noeuds des 2 parents stockés dans les 2 vecteurs de réels `t1[]` et `t2[]`

5.4 Package Tools

5.4.1 Package Tools.Tri

Contient une méthode de tri spécifique

5.4.2 Package Tools.Gen_Aleat

Contient différentes méthodes de génération de nombres aléatoires

5.5 Package Test

5.5.1 Package Test.NN_BackPropagation

Ce package contient la class `NN_BackPropagation_Test.class`. Cette class utilise la class `NN_BackPropagation.class` Elle optimise les points de réseau du neurone par la méthode de rétropropagation du gradient.

5.5.2 Package Test.Ga_NN_Opt_Learn

Ce package contient la class `Main_GA_NN_BP_Param.class`. Cette class utilise la class population `Popu1_Ga_NN_BP_Param.class` Elle optimise les deux paramètres de la rétropropagation du gradient *Epsilon* et le *Momentum*.

5.5.3 Package Test.Ga_NN_Opt_Weight

Ce package contient la class `Main_GA_NN_Davis.class`. Cette class utilise la class population `Popu1_Ga_NN_Davis.class` Elle optimise poids d'un réseau de neurones par algorithme génétique ou par algorithme hybride (génétique + backpropagation du gradient).

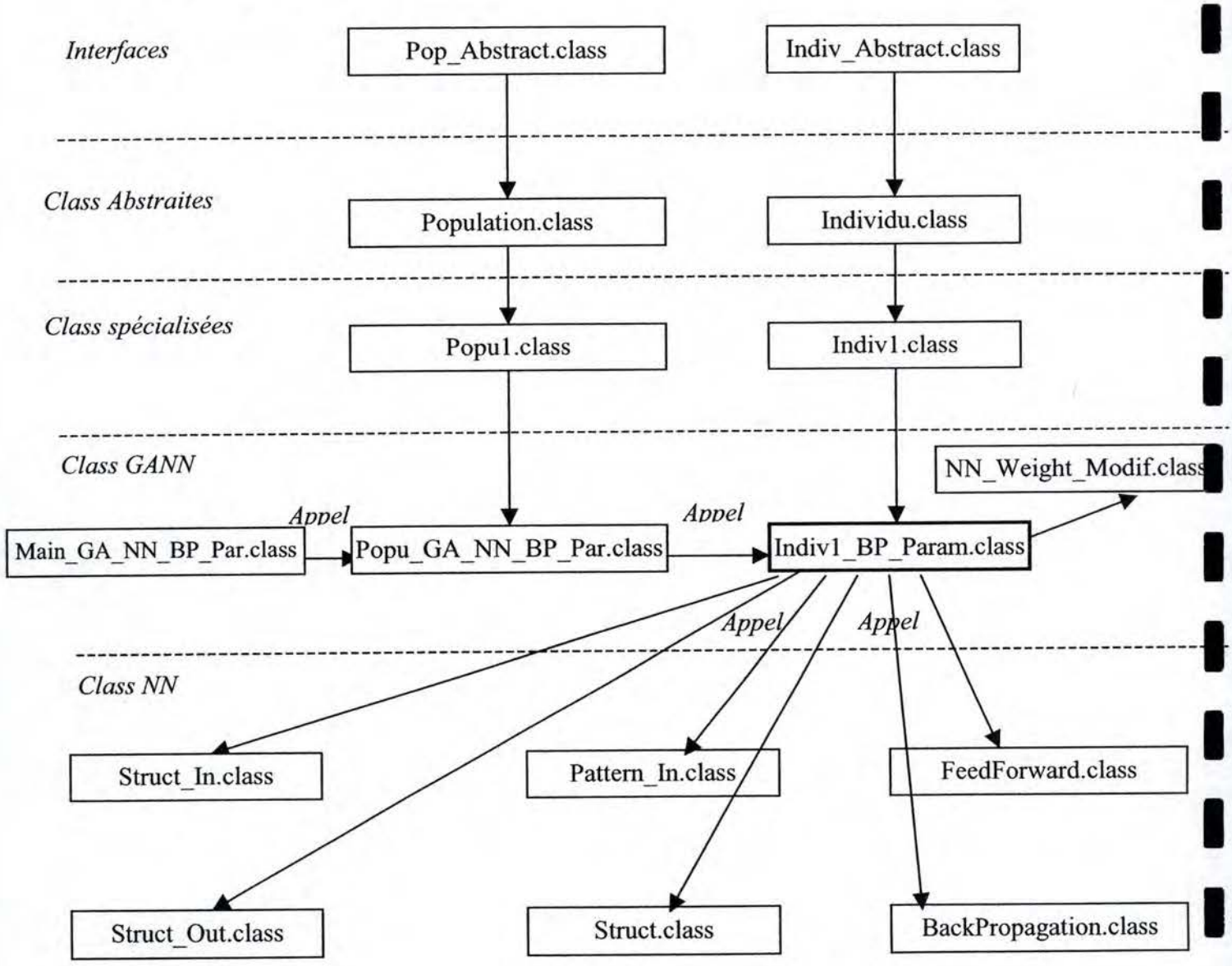
5.5.4 Package Test.Ga_NN_Opt_Struct

Ce package contient la class `Main_GA_NN_Miller.class`. Cette class utilise la class population `Popu_Bin1_Struct.class` Elle optimise la structure d'un réseau de neurones à l'aide de l'algorithme de Miller (encodage direct).

5.6 Interaction des différentes class GA et NN

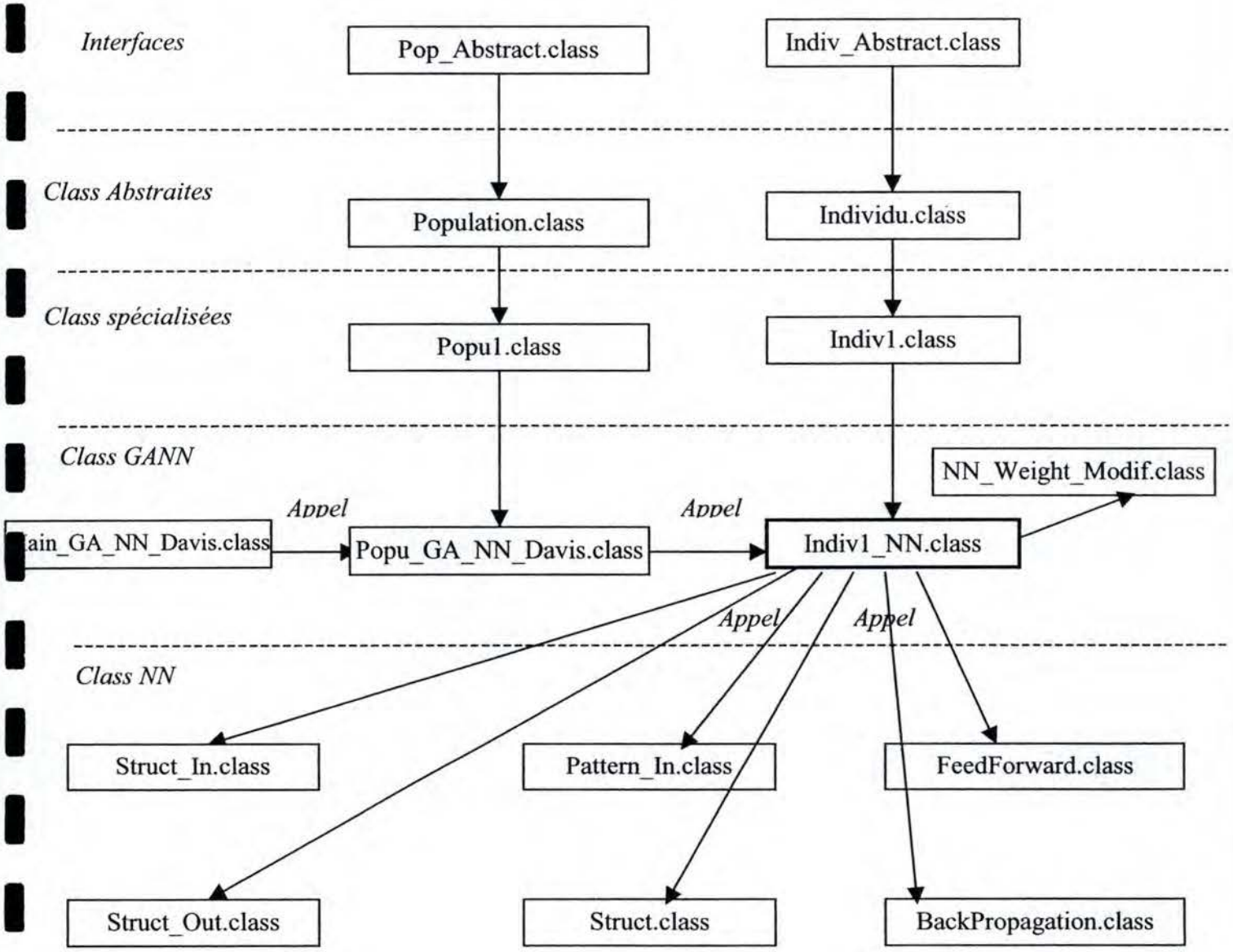
5.6.1 Configuration des différents modules optimisation des paramètres "Taux d'apprentissage" et "momentum"

Les individus encodent avec des vecteurs de réels



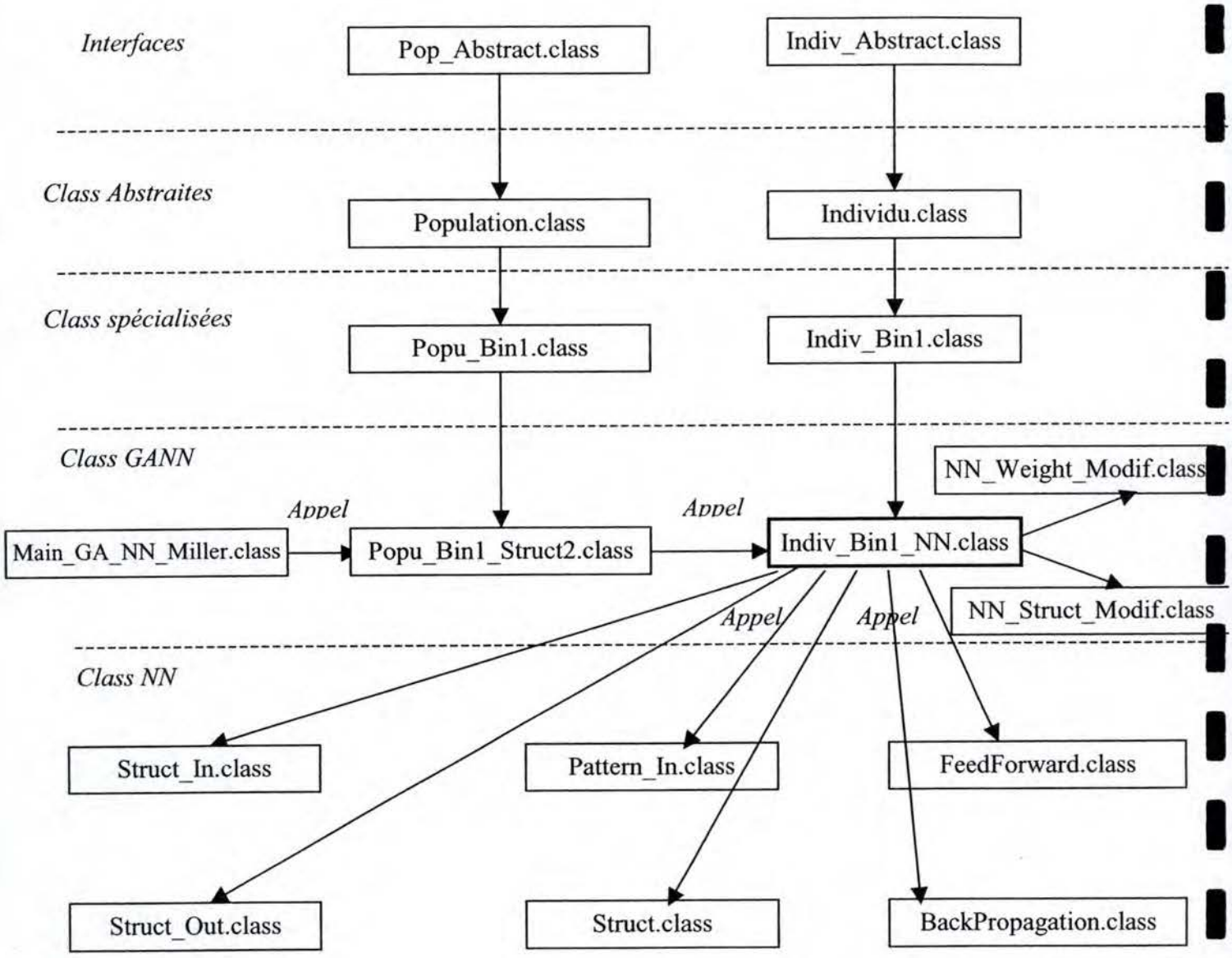
5.6.2 Configuration des différents modules optimisation des poids d'un réseau de neurones par GA

Les individus encodent avec des vecteurs de réels



5.6.3 Configuration des différents modules optimisation de structure de réseau de neurones par GA (méthode de Miller)

Les individus encodent avec des chaînes de bits



6 Chapitre 5 : Applications

6.1 Optimisation du taux d'apprentissage γ et du momentum α pour la rétropropagation du gradient BP

6.1.1 Algorithme

Cette première application qui a été exposée au point 4.4.2 du chapitre 3 (le lecteur se rapportera à ce point pour avoir plus de détails), a été modélisée par le programme d'algorithme génétique appliqué aux réseaux de neurones (le lecteur se rapportera au chapitre 4 pour plus de détails et au listing du programme en annexe).

6.1.2 Description de l'application

Cette application est une modélisation du problème du XOR. Les points des 5 éléments du groupe d'apprentissage se trouvent dans le tableau 6.1. L'architecture du réseau est représentée à la figure 6.1, c'est un réseau avec 4 neurones dont 2 neurones d'entrée, 1 neurone caché et 1 neurone de sortie.

Entrées	Sorties
1.0 1.0	0.0
0.0 1.0	1.0
1.0 0.0	1.0
0.0 0.0	0.0
0.5 0.5	0.0

Tableau 6-1: Les 5 éléments du groupe d'apprentissage pour le problème du XOR

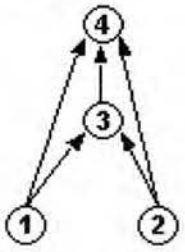


Figure 6-1: Architecture du réseau pour le problème du XOR, 2 neurones d'entrée, 1 neurone caché et 1 neurone de sortie

6.1.3 Les paramètres utilisés

Les paramètres utilisés par l'algorithme génétique sont les suivants.

- Codage : Vecteur de réels.
- Nombre d'individus: 10
- Type de croisement: "Floating Crossover" avec un taux de croisement de 0.6
- Type de mutation: "Floating Mutation" avec un taux de mutation de 0.3
- Elitisme: actif avec un taux d'élitisme de 0.85
- Scaling: actif sur base de l'ordre des fitness.

- Nombre de générations: 5000
- Fitness: obtenue par une moyenne sur le calcul de 50 réseaux dont les poids sont choisis aléatoirement entre -1 et 1.

Les paramètres utilisés pour l'apprentissage du réseau de neurones sont les suivants.

- Nombre d'itérations de la backpropagation: 20.
- Taux d'apprentissage γ : domaine compris entre 1 et 20.
- Momentum α : domaine compris entre 0 et 1.

6.1.4 Résultats

Etant donné que l'optimisation porte sur un problème avec seulement 2 variables (γ taux d'apprentissage et α Momentum) et que le domaine de ces variables est borné et continu ($1 \leq \gamma \leq 20$ et $0 \leq \alpha \leq 1$), cette application n'est pas trop ardue. Une bonne idée de la zone des bonnes solutions peut donc être obtenue, en effectuant un maillage sur le domaine, par exemple 80 x 20 points, et calculer ensuite les fitness pour ces points.

La figure 6.2 représente les solutions obtenues par ce maillage, c'est un graphique en 3 dimensions vu du dessus, les couleurs représentent les points dont la hauteur se trouve dans le même intervalle. Cette hauteur représente le pourcentage entre la Fitness au point considéré (α , γ) et la Fitness maximum obtenue avec un des points du maillage. La zone des solutions optimales est comprise entre 80 et 100 %.

Ce problème a été résolu par l'algorithme génétique 50 fois successives (points roses sur la figure 6.2) et chacune des solutions obtenues se situe dans la zone des bonnes solutions (comprise entre 80 et 100 % de la fitness maximum). L'algorithme génétique donne donc satisfaction et le calcul d'un grand nombre de points (1600 points pour un maillage 80 x 20) n'est plus nécessaire, car l'algorithme génétique converge naturellement vers la zone des bonnes solutions.

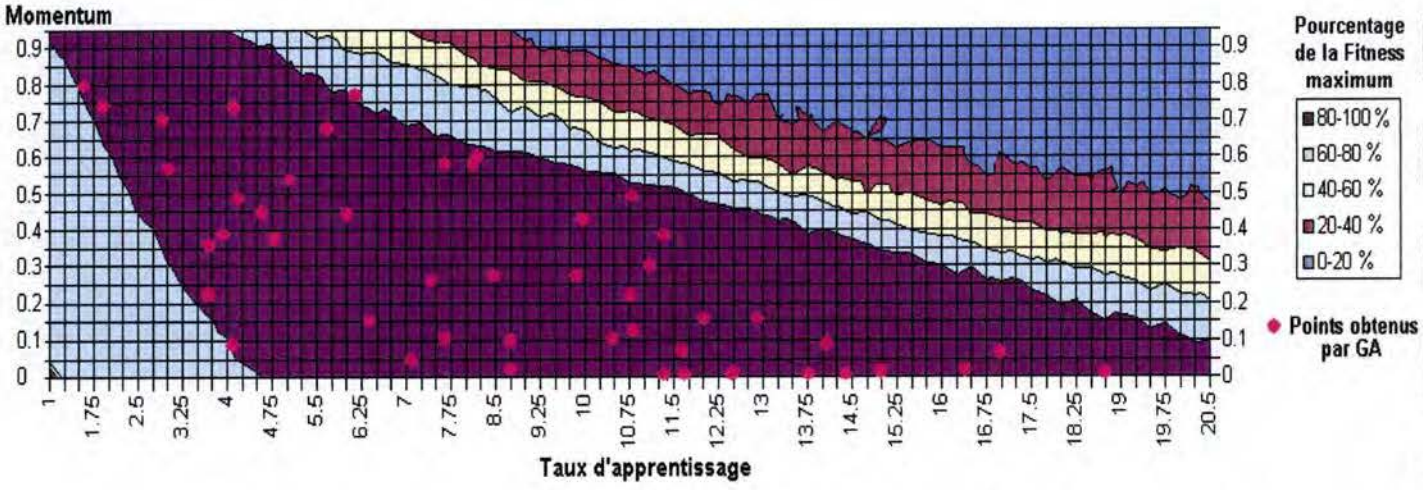


Figure 6-2 : Comparaison entre les points obtenus par l'algorithme génétique (Rose) et les valeurs obtenues par simple rétropropagation du gradient (valeurs entre 0 et 100% avec 100% la meilleure Fitness et 0% la moins bonne)

6.2 Comparaison de différentes méthodes d'apprentissage des poids de réseaux de neurones

6.2.1 Algorithme

Cette application est une comparaison portant sur les trois méthodes suivantes:

- Apprentissage par rétropropagation du gradient BP (points 3.7.1 et 3.7.2 du chapitre 2).
- Apprentissage par Algorithme génétique GA (point 4.2.6 du chapitre 3)
- Apprentissage par Algorithme hybride: GA + BP (point 4.2.7 du chapitre 3)

Le lecteur se rapportera à ces différents points pour avoir plus de détails.

6.2.2 Description de l'application

Cette application est une modélisation du problème du XOR identique au point précédent. Les points du groupe d'apprentissage se trouvent dans le tableau 6.1. L'architecture du réseau est représentée à la figure 6.1. Ce réseau a été sélectionné car il présente l'inconvénient de contenir un minimum local dans l'espace des poids du réseau [GaMa00] et [FGT92]. Lors d'une convergence de la rétropropagation du gradient on atteint ce minimum local en moyenne une fois sur deux.

6.2.3 Les paramètres utilisés

Paramètres de l'algorithme BP:

- Nombre d'itérations de la backpropagation utilisé est de 100 000.
- Taux d'apprentissage: 2.5
- Momentum: 0.4
- Poids initiaux des connexions choisis aléatoirement entre -1 et 1.

Paramètres de l'algorithme génétique:

- Codage : Vecteurs de réels
- Nombre d'individus: 200.
- Les valeurs des poids des connexions sont choisies entre -20 et 20
- Type de croisement: "Floating Crossover" avec un taux de croisement de 0.6
- Type de mutation: "Floating Mutation" avec un taux de mutation de 0.3
- Elitisme: actif avec un taux d'élitisme de 0.79
- Scaling: actif sur base de l'ordre des fitness.
- Nombre de générations: 50 000

Paramètres de l'algorithme hybride

- Codage : Vecteurs de réels
- Nombre d'itérations de la backpropagation: 100.

- Taux d'apprentissage γ : 2.5
- Momentum α : 0.4
- Poids initiaux des connexions choisis aléatoirement entre -1 et 1.
- Nombre d'individus dans la population: 200.
- Les valeurs des poids des connexions sont choisies entre -20 et 20
- Type de croisement: "Floating Crossover" avec un taux de croisement de 0.6
- Type de mutation: "Floating Mutation" avec un taux de mutation de 0.3
- Elitisme: actif avec un taux d'élitisme de 0.79
- Scaling: actif sur base de l'ordre des fitness.
- Nombre de générations: 50 000

6.2.4 Résultats

Les résultats obtenus à la figure 6.3, confirment les conclusions obtenues par Montana et Davis, lorsqu'on a des minima locaux, la méthode utilisant les algorithmes génétiques GA surpasse en moyenne la méthode classique de rétroproppagation du gradient BP. Ils confirment également les résultats obtenus par Belew, à savoir que la méthode hybride surpasse, en moyenne, non seulement la méthode classique de rétropropagation du gradiant BP, mais également la méthode d'algorithme génétique seule GA. Remarquons que ces 3 courbes sont les moyennes obtenues sur 50 essais.

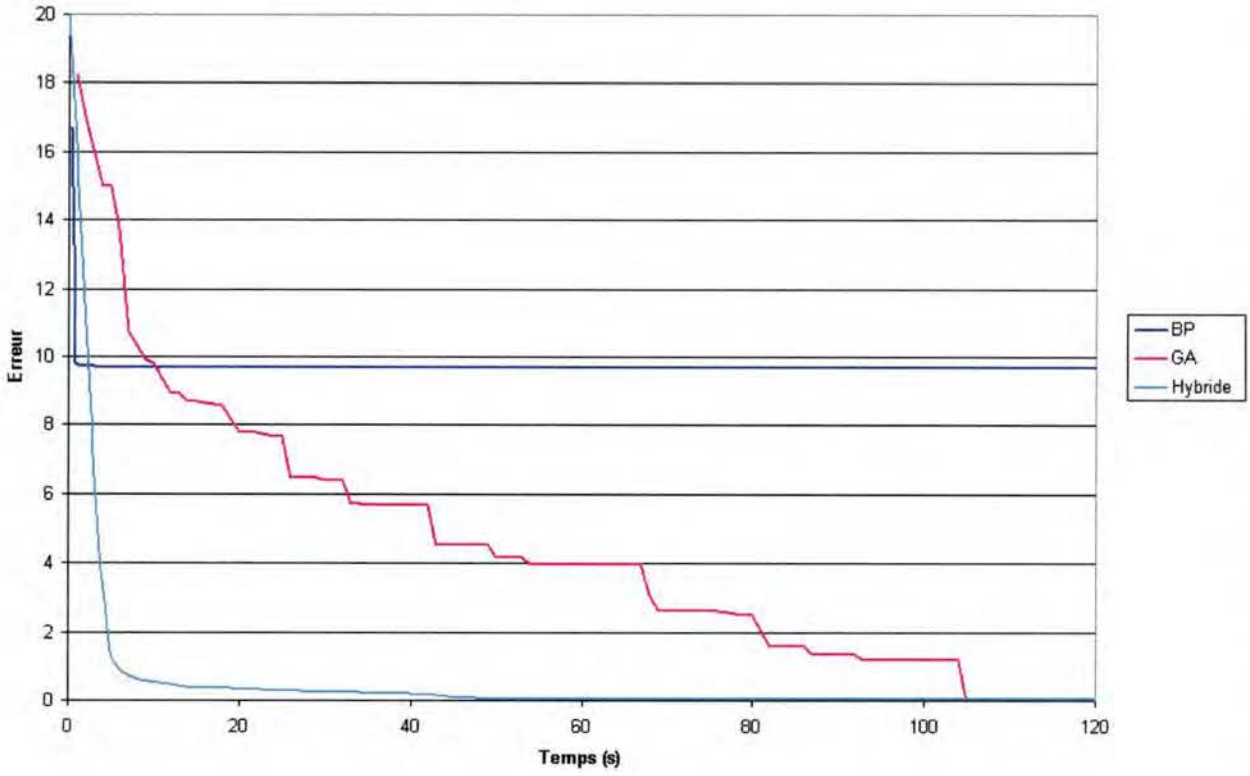


Figure 6-3: Comparaison des méthodes de rétropropagation du gradient BP, d'Algorithme génétique GA et d'Algorithme hybride, pour le problème XOR

6.3 Optimisation de structure de réseau de neurones: Méthode de Miller

6.3.1 Algorithme

Cette application se base sur l'algorithme proposé par Miller (point 4.3.2 du chapitre 3).

Comme la modélisation porte sur des réseaux Feedworward, les éléments se situant dans la partie inférieure gauche de la matrice de connexions (en dessous de la diagonale principale) voir figure 6.4, sont égales à 0 (pour rappel 1 ou 0 dans la case *ij* signifie qu'il y a ou qu'il n'y a pas, respectivement, de connexion allant du neurone *i* vers le neurone *j*).

Dans les réseaux de neurones de l'exemple de la figure 6.4, il y a deux neurones d'entrée (les valeurs des éléments de ces 2 colonnes sont égales à 0, c-à-d qu'il n'y a pas de connexion arrivant à ces neurones) et il y a un neurone de sortie (les valeurs des éléments de cette ligne sont égales à 0, c-à-d qu'il n'y a pas de connexion partant de ces neurones). Les valeurs à initialiser lors de la création des réseaux sont donc égales, respectivement pour les réseaux à 4, 5 et 6 neurones, à 5, 9 et 14 (zone en gris sur la figure 6.4) soit 2^5 , 2^9 et 2^{14} réseaux possibles. La mutation portera également sur les éléments de cette zone. Le croisement se fera par échange de colonnes.

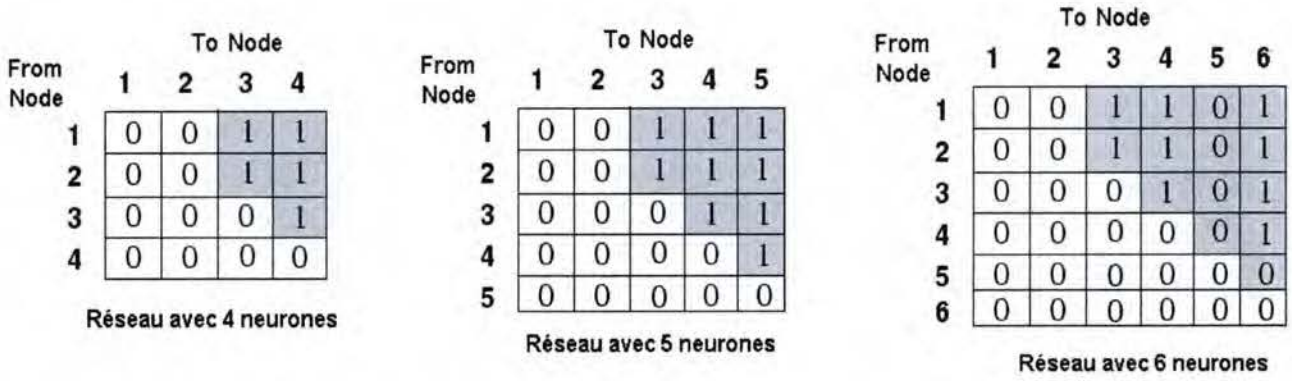


Figure 6-4: Matrice des connexions de réseaux feedforward avec 4, 5 et 6 neurones modélisant un problème à 2 neurones d'entrée et 1 neurone de sortie

L'apprentissage des poids se fera par rétropropagation du gradient pour chacun des réseaux constituant la population.

6.3.2 Description de l'application

Le réseau est codé comme mentionné au point précédent à l'aide de la méthode de l'encodage directe proposée par Miller.

Cette modélisation a été effectuée sur le réseau XOR avec 4, 5 et 6 neurones.

L'ensemble des points d'apprentissage utilisé est :

Entrées		Sorties
1.0	1.0	0.0
0.0	1.0	1.0
1.0	0.0	1.0
0.0	0.0	0.0

6.3.3 Les paramètres utilisés

Les paramètres de l'algorithme génétique qui ont été utilisés sont :

- Codage: avec des chaînes de bits
- Nombre d'individus: 15.
- Type de croisement: croisement des nœuds avec un taux de croisement de 0.6
- Type de mutation: mutation des nœuds avec un taux de mutation de 0.3
- Elitisme: actif avec un taux d'élitisme de 0.89
- Scaling: linéaire.
- Nombre de générations: 1000.

Paramètres de l'algorithme BP (utilisés pour le calcul des Fitness)

- Nombre d'itérations de la rétropropagation du gradient: 2000.
- Taux d'apprentissage γ : 0.5
- Momentum α : 0.25
- Poids initiaux des connexions choisis aléatoirement entre -1 et 1.

6.3.4 Résultats

1) Réseau obtenu avec au départ 4 neurones

L'optimisation a été effectuée successivement 10 fois pour cette configuration et les mêmes architectures de réseau ont été obtenues après convergence voir figure 6-5.

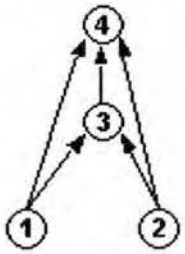


Figure 6-5 : Réseau obtenu par l'algorithme de Miller avec 4 neurones au départ

2) Réseau obtenu avec au départ 5 et 6 neurones

L'optimisation a également été effectuée successivement 10 fois pour ces 2 configurations et les mêmes architectures de réseau ont été obtenues après convergence voir figure 6-6.

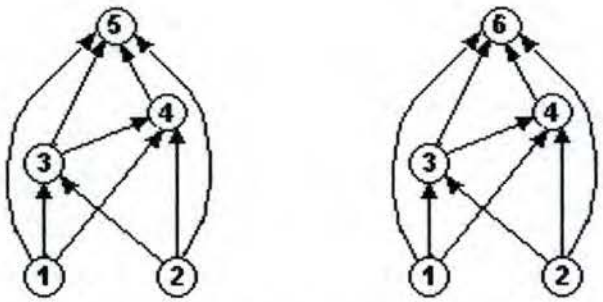


Figure 6-6 : Réseaux obtenus par l'algorithme de Miller avec 5 ou 6 neurones au départ

Il est intéressant de remarquer que les réseaux obtenus avec au départ 5 ou 6 neurones sont identiques, en effet le $5^{ème}$ neurone de l'architecture à 6 neurones est désactivé après convergence (pas de connexion aux autres neurones).

7 Conclusion

Dans la majorité des cas, le but des réseaux de neurones est de modéliser des fonctions complexes que les méthodes classiques ne peuvent modéliser simplement. Ce type de modélisation permet d'obtenir très rapidement, grâce au réseau et avec peu de calcul, les valeurs de ces fonctions complexes. Dans l'introduction, il a été montré que cette approche très séduisante ne présente qu'une seule difficulté, l'apprentissage du réseau. Ce facteur a pour conséquence que certains problèmes complexes restent sans réponse par les méthodes déterministes classiques d'apprentissage.

Cet ouvrage a permis de mettre en évidence comment les méthodes stochastiques du type algorithme génétique sont venues à bout de cette difficulté de manière très élégante. L'intégration de ces deux méthodes a donc permis de maîtriser des problèmes très complexes. Les différentes opérations permettant d'atteindre cet objectif sont:

- L'optimisation de poids des connexions, évitant de converger vers des solutions sous-optimales.
- L'optimisation de l'architecture du réseau, permettant d'obtenir un réseau avec un nombre optimal de neurones et un nombre optimal de connexions. Ce problème était ardu et purement empirique.
- L'optimisation de règles d'apprentissage, permettant de travailler avec des règles d'apprentissage intimement liées au problème traité.
- L'optimisation des entrées de réseaux, afin d'éliminer les entrées inutiles.
- L'optimisation des groupes d'éléments d'apprentissage de réseaux, pour ne garder que les éléments pertinents.

Le programme de calcul développé, nous a permis, en codant certains de ces algorithmes, de les comprendre encore plus finement et nous a montré avec quelle facilité ces algorithmes s'affranchissent des difficultés classiques liées à la modélisation des problèmes complexes.

Une perspective intéressante de continuation de ce travail, pour la partie compilation, est l'étude de la littérature relative à la résolution simultanée, pour un même problème, de plusieurs des applications décrites ci-dessus. La littérature dans ce domaine est assez abondante. Une perspective intéressante de continuation de la partie programmation, est le codage des autres algorithmes exposés dans la première partie ou le codage de nouveaux algorithmes.

8 Bibliographie

[And77] ANDERSON, J.A. : "Neural models with cognitive implications. In D. Laberge & S.J. Samuels (Eds.), Basic Process in Reading Perception and comprehension Models" (Hillsdale, NJ: Erlbaum), 1977, p 27-90

[BMIS90] BELEW K. McINERNEY J. SCHRAUDOLPH N. : "Evolving networks: Using the Genetic Algorithm with Connectionist Learning" (Cognitive Computer Science Research Group). University California at San Diego La Jolla CA 92093 , CSE Technical Report # CS90-174, June 1990

[Boer92] BOERS E. and KUIPER H. : "Biological metaphors and the design of modular artificial neural networks." (Master's Thesis). Leiden University, The Netherlands, 1992

[Cer94] CERF, R. : "Une théorie asymptotique des algorithmes génétiques" (Thèse, Université de Montpellier II), 1994

[Cha90] CHALMERS D. "The evolution of learning : An experiment in genetic connectionism." (In D.S. Touretzky, J.L. Elman, T.J. Sejnowski and G.E. Hinton eds - Proceedings of the 1990 Connectionist Models Summer School - Morgan Kaufmann V), 1990, p 1-20

[ChLp91] CHANG E. LIPPMANN P. : "Using Genetic Algorithms to improve Pattern Classification Performance" (Lincoln Laboratory, MIT Lexington, MA 02173-9108), 1991, p797-803.

[ChSe95] CHINRUNGRUENG C., SEQUIN C.H. : "Optimal adaptive k-means algorithm with dynamic adjustment of learning rate" (IEEE Transaction on Neural Networks, 6(1)), 1995, p157-169.

[CPN95] CANGELOSI A., PARISI D. & NOLFI S. : "Cell division and migration in 'genotype' for neural networks" (Network: computation in neural systems), 1995.

[Cun85] CUN, Y.L. : "Une procédure d'apprentissage pour réseau à seuil assymetrique" (Proceeding of cognitiva. 85, 599-604), 1985

[Dav87] DAVIS, L. : "Genetic Algorithms and Simulated Annealing" (Morgan Kaufmann Publishers), 1987

[FGT92] FRASCONI, P., GORI M., TESI A. : "Successes and failures of backpropagation: a theoretical investigation", Technical Report, Universita di Firenze, Dipartimento di Sistemi e Informatica, Via di Sant Marta 3, 1992, pp23

[GaMa00] GALLAGHER M. R. : "Multi-Layer Perceptron Error Surfaces: Visualization, Structure and Modelling" (PhD Thesis, University of Queensland), 2000, p56-61.

[Gol89] GOLDBERG, D.E. : "Genetic Algorithms in search, optimisation and Machine Learning" (Addison-Wesley), 1989

[GoRi87] GOLDBERG, D.E., RICHARDSON, J.J. : "Genetic Algorithms with sharing for multimodal function optimisation", (Proceedings of the Second International Conference on Genetic Algorithms ICGA), 1987

[GeYi93] GERMAY, N., XIADONG, Y. : "A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimisation", (Rapport technique, Université Catholique de Louvain, Laboratoire d'Electronique et d'instrumentation)

[Gro76] GROSSBERG, S. : "Adaptative pattern classification and universal recoding I&II" (Biological Cybernetics. 23), 1976, p121-134 p 187-202

[Gru92] GRUAU, F. "Genetic synthesis of Boolean neural networks with a cell rewriting developmental process", (in L. D. Whitley and J.D. Schaffer, eds, COGANN-92 International Workshop on Combinations of genetic Algorithms and Neural Networks. IEEE Computer Society Press), 1992

[GuUh92] GUO Z. & UHRIG R. "Using Genetic Algorithms to Select Inputs for Neural Networks", (in L. D. Whitley and J.D. Schaffer, eds, COGANN-92 International Workshop on Combinations of genetic Algorithms and Neural Networks. IEEE Computer Society Press), p223-234, 1992

[Harp89] HARP S., SAMAD T., GUHA A. : "Toward the genetic synthesis of neural networks." (in D.J. Schaffer, editor, 3rd intern. Conf. On Genetic Algorithms), page 360-369, 1989

[Harp90] HARP S., SAMAD T., GUHA A. : "Designing application-specific neural networks using the genetic algorithm." (in D.S. Touretsky (Ed.), advnaces in neural information processing 2), San Mateo, CA: Morgan Kaufmann, page 447-454, 1990

[Hol62] HOLLAND J.H. : "Outline for a logical theory of adaptive systems" (Journal for the Association of Computing Machinery. 3), 1962

[Heb49] HEBB D.O. : "The organisation of Behaviour." (New York: Wiley), 1949

[Hol75] HOLLAND J.H. : "Adaptation in Natural and Artificial Systems" (University of Michigan press), 1975

[Hol75] HOLLAND J.H. : "Adaptation in Natural and Artificial Systems" (University of Michigan press), 1975

[Hop82] HOPFIELD J. : "Neural networks and physical systems with emergent collective computational abilities" (Proceedings of the National Academy of Sciences, 79, 2554-2558), 1982

[Kita90] KITANO, H, "Design neural networks using genetic algorithms with graph generation system" (complex Systems 4 : 461-476 V), 1990

[Kita94] KITANO, H "Neurogenetic learning: in integrated method of designing and training neural networks using genetic algorithms" Physica D 75 : 225-238 V, 1994

[KMMR96] KOK J.N., MARCHIORI E., MARCHIORI M. & ROSSI C. "Constraining of Weights using Regularities", 1-6 , 1996

[KMMR97] KOK J.N., MARCHIORI E., MARCHIORI M. & ROSSI C. "Evolving Training of CLP-Constrained Neural Networks", 1-14 , 1997

[Koh77] KOHONEN, T., : "Associative Memory: A System-Theorical Approach." (Springer-Verlag), 1977

[Koz90] KOZA J., : "Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems" (Technical Report STAN-CS-90-1314, Depertement of computer Science, Stanford University), Stanford, CA, 1990

[Koz91] KOZA J., RICE J.: "Genetic generation of both the weight and architecture for a neural network." (In IJCNN-91-Seattle: International Joint conference on Neural Networks, volume 2, page 397-404, Seattle, Washington, USA, IEEE Press, 8-14 July 1991

[Koz92] KOZA J. : "Genetic programming: on the Programming of compuers by Means of natural Selection." (The MIT Press, Cambridge, MA), 1992

[KrCi94] KRISHMAN R. et CIESIELSKI V. : "2Delta-GANN: A new approach to training neural networks using genetic algorithms." (in A.C. Tsoi [Ed.] Proceeding of the fifth australian conference on enural networks, University of Queensland), Brisbane p38-41, 1994

[MaGo93] MAHFOUD, S. W., GOLDBERG, D. E. : "Parallel recombinative simulated anealing: A genetic algorithm" (Parallel computing, 21), 1993, p 1-28

[MaGo93] MAHFOUD, S. W., GOLDBERG, D. E. : "Parallel recombinative simulated anealing: A genetic algorithm" (Parallel computing, 21), 1993, p 1-28

[MCPi43] McCULLOCH, W.S., PITTS W. : "A logical calculus of ideas immanent in nervous activity" (*Bulletin of Mathematical Biophysics, 5*), 1943, p 115-133

[Mil89] MILLER, G, TODD, P., & HEGDE S., "Designing neural networks using genetic algorithms" (in J.D. Schaffer ed Proceedings of the third international conference on genetic algorithms - Morgan Kaufmann), 1989

[MiPa69] MINSKY M. , PAPERT S. : "Perceptrons: An Introduction to Computational Geometry" (MIT Press), 1969

[MoDa89] MONTANA D.J., DAVIS L. : "Training feedforward networks using genetic algorithms" (In Proceedings of the international Joint Conference on artifical intelligence - Morgan Kaufmann), 1989

[NgLi91] NG K. & LIPPMANN P. : "A comparative Study of the pratical Characteristic of Neural Networks and Conventional Pattern Classifiers" (In Lippmann, R. Moody and J. Touretzky, Eds Advances in Neural Information Processing Systems 3, 1991

[RiLi96] RICHARD & LIPPMANN : "Back Propagation Family Album" (Department of Computing Macquarie University NSW 2109, Technical Report C/TR96-05) august 1996

[Ros59] ROSENBLATT, F. : "Principles of Neurodynamics" (New York: Spartan Books), 1959

[Rum86] RUMELHART, D. E., HINTON, G.E. & WILLIAMS R.J. : "Learning representation by backpropagation errors" (Nature, 323, 533-536), 1986

[Sch92] SCHAFFER & WHITLEY, "Combinations of genetic algorithms and neural networks: a survey of the state of the art", (in L. D. Whitley and J.D. Schaffer, eds, COGANN-92 International Workshop on Combinations of genetic Algorithms and Neural Networks. IEEE Computer Society Press V), 1992

[SDG98] SEXTON S., DORSEY E. & JOHNSON J. : "Toward global optimization of neural networks: a comparison of the genetic algorithm and backpropagation" (University of mississippi), 1998

[WeSuTh95] WELLER P.R., SUMMERS R. & THOMPSON A.C. : "Using a Genetic Algorithm to evolve an Optimum Input Set for a Predictive Neural Network" (Genetic Algorithms in Engineering Systems: Innovations and applications 12-14, Conference Publication N° 414), Sept 1995

Annexes

9 Annexes

9.1 Listing des différents programmes Java

9.1.1 Individu

9.1.1.1 Indiv_Abstract

```
package Ga.Individu;

public interface Indiv_Abstract {

    void Indiv_in_Fit_Scl(double g);
    void Indiv_in_Fit_Norma(double d);
    double Indiv_Fitness();
    double Indiv_Fitness_Scl();
    double Indiv_Fitness_Norma();
    boolean Indiv_Out_Bound();

    void Indiv_Eval();
    void Indiv_Affich();
    int Nbr_Chrom_Get();
    int Nbr_Locus_Get();
    Object clonne();

    void Chromo_in(int i, Object d);
    Object Chromo_Val(int i);
    Object Chromo_Gen_Aleat(int i);

    public boolean Get_Indiv_New();
    public void Set_Indiv_New(boolean flag);

}
```

9.1.1.2 Individu

```
package Ga.Individu;
import Tools.Gen_Aleat.Gen_Aleat; // pour générer des nombres
aléatoires
import java.lang.Math; // pour les fonctions mathématiques

public abstract class Individu implements Indiv_Abstract {

protected int  Nbr_Chrom; // Nombre de Chromosomes dans l'individu
protected int  Nbr_Locus; // Nombre d'éléments dans chaque chromosome
protected boolean Indiv_New = false; //Indique si l'individu a déjà
été évalué
protected double Fitness;  //  Fitness de l'individu
protected double Fitness_Scaled;  // Fitness de l'individu après
Scaling
protected double Fitness_Norma;  // Fitness de l'individu après
normalisation (de 0 à 1)
protected boolean Bound_Actif; // Indique si les valeurs des
chromosomes sont dans un domaine borné
protected boolean Bound_Check = true; // Indique si les valeurs des
chromosomes sont comprises dans le domaine
```

```
protected Gen_Aleat g = new Gen_Aleat(); // élément pour génération
aléatoire

/*************************************************/
public void Indiv_in_Fit_Scl(double d)
// Remplace par d la valeur de la variable "Fitness_Scales" de
l'individu
{ Fitness_Scaled = d; }

/*************************************************/
public void Indiv_in_Fit_Norma(double d)
// Remplace par d la valeur de la variable "Fitness_Norma" de
l'individu
{ Fitness_Norma = d; }

/*************************************************/
public double Indiv_Fitness()
// Renvoit la valeur de la variable "Fitness"
{ return Fitness; }

/*************************************************/
public double Indiv_Fitness_Scl()
// Renvoit la valeur de la variable "Fitness_Scaled"
{ return Fitness_Scaled ; }

/*************************************************/
public double Indiv_Fitness_Norma()
// Renvoit la valeur de la variable "Fitness_Norma"
{ return Fitness_Norma; }

/*************************************************/
public int Nbr_Chrom_Get()
{ return Nbr_Chrom ; }

/*************************************************/
public int Nbr_Locus_Get()
// Renvoit la valeur de la variable "Nbr_Locus"
{ return Nbr_Locus ; }

/*************************************************/
public boolean Indiv_Out_Bound()
// renvoit un booléan indiquant si le domaine des valeurs des
chromosomes
// possède des bornes supérieures et inférieures
{ return true; }

/*************************************************/
public boolean Get_Indiv_New()
// Renvoit la valeur de la variable "Indiv_New"
{ return Indiv_New; }

/*************************************************/
public void Set_Indiv_New(boolean flag)
// Remplace par flag la valeur de la variable "Indiv_New" de
l'individu
{ Indiv_New = flag; }

}
```

9.1.1.3 Indiv1

```
package Ga.Individu;
public class Indiv1 extends Individu {

protected double [] Chrom;
protected double [] Low_Bound;
protected double [] High_Bound;

/************************************************/
/************************************************/
public  Indiv1()
// Constructeur
{
  Nbr_Chrom = 2;
  Nbr_Locus = 1;
  Chrom = new double[Nbr_Chrom];
  Low_Bound = new double[Nbr_Chrom];
  High_Bound = new double[Nbr_Chrom];
  Bound_Actif = true;

  int i;
  for(i=0 ; i<Nbr_Chrom ; i++)
    { Low_Bound[i] = 0.0;
      High_Bound[i] = 1.0;
      Double ObjDbl = (Double)Chromo_Gen_Aleat(i);
      Chrom[i] = ObjDbl.doubleValue();
    }
}

/************************************************/
public Object Chromo_Gen_Aleat(int i)
// Génère aléatoirement un nouveau chromosme "réel" entre les bornes
inférieures
// et supérieures du chromosome
{
      double r = g.Gen_Aleat_Double(Low_Bound[i] , High_Bound[i] ) ;
        Double ObjDbl = new Double(r);
      return ObjDbl;
}

/************************************************/
public Object clonne()
//  Clone l'individu et renvoit ce nouvel individu
{
      Indiv1 i = new Indiv1();
      i.Nbr_Chrom = this.Nbr_Chrom;
      i.Indiv_New = this.Indiv_New;
      i.Fitness = this.Fitness;
      i.Fitness_Scaled = this.Fitness_Scaled;
      i.Fitness_Norma = this.Fitness_Norma;

      for(int j=0 ; j<Nbr_Chrom ; j++) {
            i.Chrom[j] =this.Chrom[j];
            i.Low_Bound[j] =this.Low_Bound[j];
            i.High_Bound[j] =this.High_Bound[j];
      }
      return i;
}
```

```
/************************************************/
public void Bound_Eval()
// Evalue si les chromosome de l'individu sont entre ses bornes
respectives
{
  if (Bound_Actif == true)
  {
  for(int i=0 ; i<Nbr_Chrom ; i++)
    {  if (Chrom[i] < Low_Bound[i] ) { Bound_Check = false ; };
       if (Chrom[i] > High_Bound[i] ) { Bound_Check = false; };
    }
  }
}

/************************************************/
public void Indiv_Affich()
// Méthode affichant l'individu
{
  int i;
  for(i=0 ; i<Nbr_Chrom  ; i++)
  {
    System.out.print("\tchrom" + i + " " + Chrom[i] ) ;
   }
    System.out.print("\tFit " + Fitness ) ;
    System.out.print("\tFitScl " + Fitness_Scaled ) ;
    System.out.println("\tFitNorm " + Fitness_Norma ) ;
}


/************************************************/
public Object Chromo_Val(int i)
// renvoit la valeur de chromosome i
{
   Double d =  new Double(Chrom[i]);
   return d;
}

/************************************************/
public void Chromo_in(int i, Object d)
// Remplace le chromosome i par l'objet d
{
   Double ObjDbl = (Double)d;
   Chrom[i] = ObjDbl.doubleValue();
}

/************************************************/
public void Indiv_Eval()
// évaluation de l'individu
{
   if(Bound_Check == true)
   {
     Fitness = Math.sin( Chrom[0]*19.0)* Math.cos( Chrom[1] *11.0) +
Chrom[0]*3*Chrom[1];
   }
   else
   {
       Fitness = 0;
   }
     Indiv_New = true;
}
}
```

9.1.1.4 Indiv_Bin1

```
package Ga.Individu;
import Tools.Gen_Aleat.Gen_Aleat;
import java.lang.Math;

public class Indiv_Bin1 extends Individu {

protected int Nbr_Tot;
public int Chrom_Bin[];

/*************************************************/
/*************************************************/
public Indiv_Bin1()
// Constructeur
{
  Nbr_Chrom = 1;
  Nbr_Locus = 30;
  Nbr_Tot = Nbr_Chrom * Nbr_Locus;
  Chrom_Bin  = new int[Nbr_Tot];
  Bound_Actif = true;

  for(int i=0 ; i<Nbr_Tot ; i++)
    { double Aleat_Bin = g.Gen_Aleat_Double(0, 1) ;
        if (Aleat_Bin<0.5) {   Chrom_Bin[i]=0 ; }
      else  {   Chrom_Bin[i]=1 ; };
    }
}

/*************************************************/
/*************************************************/
public Indiv_Bin1(int Nbr_Chrom1, int Nbr_Locus1)
//  Constructeur en spécifiant le nombre de chromosomes "Nbr_Chrom1"
et
//  en spécifiant le nombre de bits par chromosome "Nbr_Locus1"
{
  Nbr_Chrom = Nbr_Chrom1;
  Nbr_Locus = Nbr_Locus1;
  Nbr_Tot = Nbr_Chrom1 * Nbr_Locus1;
  Chrom_Bin  = new int[Nbr_Tot];

  for(int i=0 ; i<Nbr_Tot ; i++)
    { double Aleat_Bin = g.Gen_Aleat_Double(0, 1) ;
        if (Aleat_Bin<0.5) {   Chrom_Bin[i]=0 ; }
      else  {   Chrom_Bin[i]=1 ; };
    }
}

/*************************************************/
public Object clonne()
//  Clone l'individu et le renvoit

{
      Indiv_Bin1 i = new Indiv_Bin1();
      i.Nbr_Chrom = this.Nbr_Chrom;
        i.Nbr_Locus = this.Nbr_Locus;
        i.Nbr_Tot = this.Nbr_Tot;

      i.Indiv_New = this.Indiv_New;
      i.Fitness = this.Fitness;
```

```
        i.Fitness_Scaled = this.Fitness_Scaled;
        i.Fitness_Norma = this.Fitness_Norma;

        for(int j=1 ; j<Nbr_Tot ; j++) {
              i.Chrom_Bin[j] = Chrom_Bin[j];
        }
        return i;
}

/*************************************************/
public void Indiv_Affich()
//  Affiche l'individu
{
  System.out.print(" ");
  for(int i=0 ; i<Nbr_Tot ; i++) {
    System.out.print(" " + Chrom_Bin[i]);
  }
    System.out.print("\tFit " + Fitness ) ;
    System.out.print("\tFitScl " + Fitness_Scaled ) ;
    System.out.println("\tFitNorm " + Fitness_Norma ) ;

}

/*************************************************/
public Object Chromo_Val(int j)
// renvoit la valeur de chromosome i
{
    int u = Chrom_Bin[j];
    Integer ObjInt = new Integer(u);
    return ObjInt;
}


/*************************************************/
public void Chromo_in(int i, Object b)
// Remplace le chromosome i par l'objet b
{
      Integer ObjInt = (Integer)b;
     Chrom_Bin[i] = ObjInt.intValue();
}

/*************************************************/
public Object Chromo_Gen_Aleat(int i)
// Génère aléatoirement un nouveau chromosme bianire 1 ou 0
{
        int r = (int)g.Gen_Aleat_Double(0, 2) ;
      Integer ObjInt = new Integer(r);
        return ObjInt;
}

/*************************************************/
public void Indiv_Eval()
// évaluation de l'individu
{

   //partie decode
   double accum = 0.0;
   double powerof2 = 1.0;
   for(int j=0; j < (Nbr_Locus); ++j)
   {
      int k = Chrom_Bin[j];
```

```
        if (k==1) { accum = accum + powerof2;} ;
        powerof2 = powerof2 *2;
    }
    double x=accum;

    //partie eval
    double coef = 1073741823.0;
    int n = 10;
    x = x / coef;
    int y=n;
    Fitness = Math.exp( y* Math.log(x) ) ;

}

}
```

9.1.1.5 Indiv_Bin2

```
package Ga.Individu;
import Tools.Gen_Aleat.Gen_Aleat;
import java.lang.Math;

public class Indiv_Bin2 extends Indiv_Bin1 {

/************************************************/
/************************************************/
public Indiv_Bin2()  // spécifique
{
  super(2, 15);
 }

/************************************************/
public void Indiv_Eval() // spécifique
{

    //partie decode
    double accum = 0.0;
    double powerof2 = 1.0;
    double []x;
    x = new double[2];
    for (int u=0; u < (Nbr_Chrom) ; ++u)
     {
      for(int j=0; j < (Nbr_Locus); ++j)
        {
         int k = Chrom_Bin[j];
           if (k==1) { accum = accum +     powerof2;} ;
           powerof2 = powerof2 *2;
         }
    x[u] = accum/(16384*2);
//     System.out.print("\t x " + x[u] ) ;
    }

    //partie eval
    Fitness = Math.sin( x[0]*19.0)* Math.cos( x[1] *11.0) +
x[0]*3*x[1];
}

}
```

9.1.1.6 Indiv1_BP_Param

```
package Ga.Individu;
import Tools.Gen_Aleat.Gen_Aleat;
import java.lang.Math;
import NN.Struct_In.*;
import NN.Struct.*;
import NN.FeedForward.*;
import NN.BackPropagation.*;
import NN.Pattern.*;
import Ga_Nn.Opt_Weight.*;
import java.io.*;

/************************************************/
public class Indiv1_BP_Param extends Indiv1 {

private int Evaluation;
private double [] Chrom;
private double [] Low_Bound;
private double [] High_Bound;
public NN_Struct NN ;
NN_Pattern_In Data_Out;
NN_FeedForward Data0;
NN_BackPropagation Data1;

   private static String title = "Xor_v02";
   private static int iter_BP = 20;
   private static int iter_j = 50; //50
      //String title = "Xor_v01";
      //String title = "Xor_v02";
      //String title = "Cancer_v01";
      //String title = "Cancer_v02";
      //String title = "Cancer_v03";
      //String title = "X1_Mult_X2_v01";
      //String title = "Symetrie_v01";
        //String title = "Symetrie_v02";

/************************************************/
public NN_Struct get_struct()
//  Renvoit la structure du réseau de neurones
{
      return NN;
}

/************************************************/
public void Set_Param(int New_iter_j , int New_iter_BP, String
New_title)  {

   iter_BP =  New_iter_BP;
   title =  New_title;
   iter_j =  New_iter_j ;

}

/************************************************/
public Indiv1_BP_Param()
// Constructeur
{

      try
```

```
      {
      String Nomfich = "Samples/"  + title  +
"/Weight_Struct_In.txt";
      NN_Weight_Struct_In  NN_In = new NN_Weight_Struct_In(Nomfich);
        NN = NN_In.get_Struct();

        String File_Data = "Samples/"  + title  + "/Pattern.txt";
      Data_Out = new NN_Pattern_In(File_Data);
        Data0 = new NN_FeedForward();
        Data1 = new NN_BackPropagation(NN);
      }
        catch (IOException e)
      {
      }

  Nbr_Chrom = 2;
  Nbr_Locus = 1;
  Chrom = new double[Nbr_Chrom];
  Low_Bound = new double[Nbr_Chrom];
  High_Bound = new double[Nbr_Chrom];

  Low_Bound[0] = 0.0; High_Bound[0] = 20.0;
  Low_Bound[1] = 0.0; High_Bound[1] = 1.0;

  for(int i=0; i<Nbr_Chrom ; i++)
    {
        Chrom[i] = g.Gen_Aleat_Double(  Low_Bound[i] , High_Bound[i]
) ;
    }
}

/*************************************************/
public Object clonne()
//  Clone l'individu et le renvoit

{
      Indiv1_BP_Param i = new Indiv1_BP_Param();
      i.Nbr_Chrom = this.Nbr_Chrom;
      i.Indiv_New = this.Indiv_New;
      i.Evaluation = this.Evaluation ;
      i.Fitness = this.Fitness;
      i.Fitness_Scaled = this.Fitness_Scaled;
      i.Fitness_Norma = this.Fitness_Norma;

      for(int j=0 ; j<Nbr_Chrom ; j++) {
            i.Chrom[j] =this.Chrom[j];
            i.Low_Bound[j] =this.Low_Bound[j];
            i.High_Bound[j] =this.High_Bound[j];
      }
      return i;
}

/*************************************************/
public void Indiv_Affich()
//  Affiche l'individu
{

  for(int i=0; i<Nbr_Chrom ; i++)
    { System.out.print("  c" + i + "   " + Chrom[i]  );
    }
```

```
    System.out.print("\tFit " + Fitness ) ;
    System.out.print("\tFitScl " + Fitness_Scaled ) ;
    System.out.println("\tFitNorm " + Fitness_Norma ) ;
}

/************************************************/
public void Indiv_Eval()
// évaluation de l'individu
{
 if(Indiv_New == false)
 {
   double s = 0;
   double err=0;
   double err_tot=0;

   int Nbr_Indiv = Data_Out.Get_Nbr_Sample();
   int Nbr_Layer = NN.Get_Nbr_Layer();

   Data1.Set_Epsilon( Chrom[0] );
   Data1.Set_Momentum( Chrom[1] );

//   System.out.print(" Chrom[0]   "+ Chrom[0] );
//   System.out.print(" Chrom[1]   "+ Chrom[1] );

   NN_Weight_Modif NN_W;
   int Nbr = NN.Get_Nbr_Weight() + NN.Get_Nbr_Neur();
   double [] Val = new double[Nbr];

//   System.out.println(" nbr "+ Nbr);

   for(int j=1; j < (iter_j+1) ; ++j)
   {
     for(int i=0; i<Nbr ; i++)
       {  Val[i] = g.Gen_Aleat_Double(-1 , 1 ) ;
//                      System.out.println(" " + Val[i] );
       }

     NN_W = new NN_Weight_Modif(NN, Val);
     s=0;

     for(int i=1; i<iter_BP ; ++i)
     {  err=0;
        for(int u=1;u< (Nbr_Indiv+1); ++u)
        {
           Data0.UpDateInput(NN, Data_Out, u);
         for (int h=2; h<(Nbr_Layer+1);++h)
               {  Data0.UpDateLayer(NN,h);   }
           err = Data0.LessSqrError(NN, Data_Out, u)+err;
           Data1.UpDateWeightandThresh(Data_Out, u);  //
        }
       s++;
       err = 100* err / Nbr_Indiv;
     }
     err_tot = err + err_tot;
   }
   err_tot = err_tot / iter_j ;
   Fitness =   100-err_tot ;
   }
   Indiv_New = false;
}
}
```

9.1.1.7 Indiv1_NN

```
package Ga.Individu;
import Tools.Gen_Aleat.Gen_Aleat;
import java.lang.Math;
import NN.Struct_In.*;
import NN.Struct_Out.*;
import NN.Struct.*;
import NN.FeedForward.*;
import NN.BackPropagation.*;
import NN.Pattern.*;
import Ga_Nn.Opt_Weight.*;
import java.io.*;


public class Indiv1_NN extends Indiv1 {

private int Evaluation;

   private static boolean flag_BP = false;
   private static String title = "Xor_v02";
   private static int iter_BP = 100;
      //String title = "Xor_v01";
      //String title = "Cancer_v01";
      //String title = "Cancer_v02";
      //String title = "Cancer_v03";
      //String title = "X1_Mult_X2_v01";
      //String title = "Symetrie_v01";

    public NN_Struct NN;
    NN_Pattern_In Data_Out;
    NN_FeedForward Data0;

   double s = 0;
   double err=0;


/*************************************************/
public NN_Struct get_struct()
//  Renvoit la structure du réseau de neurones
{
      return NN;
}

/*************************************************/
public void Set_Param(boolean New_flag_BP, int New_iter_BP, String
New_title)  {

   iter_BP =  New_iter_BP;
   title =  New_title;
   flag_BP =  New_flag_BP;

}

/*************************************************/
public Indiv1_NN()
// Constrcuteur
{
      try
      {
```

```
        String Nomfich = "Samples/"  + title  + "/Struct_in.txt";
         NN_Struct_In  NN_In = new NN_Struct_In(Nomfich);
           NN = NN_In.get_Struct();

         String File_Data = "Samples/"  + title  + "/Pattern.txt";

         Data_Out = new NN_Pattern_In(File_Data);
         Data0 = new NN_FeedForward();
      }
        catch (IOException e)
      {
      }

  Nbr_Chrom = NN.Get_Nbr_Weight() + NN.Get_Nbr_Neur();
  Nbr_Locus = 1;
  Chrom = new double[Nbr_Chrom];
  Low_Bound = new double[Nbr_Chrom];
  High_Bound = new double[Nbr_Chrom];

  int i;
  for(i=0; i<Nbr_Chrom ; i++)
    { Low_Bound[i] = -20.0; High_Bound[i] = 20.0;
        Chrom[i] = g.Gen_Aleat_Double(-20 , 20 ) ;
    }
}

/************************************************/
public Object clonne()
//public Indiv_Abstract clone()

{
      Indiv1_NN i = new Indiv1_NN();
      i.Nbr_Chrom = this.Nbr_Chrom;
      i.Indiv_New = this.Indiv_New;
      i.Evaluation = this.Evaluation ;
      i.Fitness = this.Fitness;
      i.Fitness_Scaled = this.Fitness_Scaled;
      i.Fitness_Norma = this.Fitness_Norma;

      for(int j=0 ; j<Nbr_Chrom ; j++) {
            i.Chrom[j] =this.Chrom[j];
            i.Low_Bound[j] =this.Low_Bound[j];
            i.High_Bound[j] =this.High_Bound[j];
      }
      return i;
}

/************************************************/
public void Indiv_Affich()
//  Affiche l'individu
{
  for(int i=0; i<Nbr_Chrom ; i++)
    { System.out.print("  " + Chrom[i]  );
    }

    System.out.print("\tFit " + Fitness ) ;
    System.out.print("\tFitScl " + Fitness_Scaled ) ;
    System.out.println("\tFitNorm " + Fitness_Norma ) ;
}
```

```
/************************************************/
public void Indiv_Eval()
// évaluation de l'individu
{
        int Nbr_Indiv = Data_Out.Get_Nbr_Sample();
        int Nbr_Layer = NN.Get_Nbr_Layer();

        NN_Weight_Modif NN_W = new NN_Weight_Modif(NN, Chrom);

        //Algorithme hybride GA + BP
        if(flag_BP ==true)
        {
           for(int i=1; i<iter_BP; ++i)    //
           {                               //
           err=0;
           for(int u=1;u< (Nbr_Indiv+1); ++u)
            {
               Data0.UpDateInput(NN, Data_Out, u);
             for (int h=2; h<(Nbr_Layer+1);++h)
             {
                   Data0.UpDateLayer(NN,h);
             }
                 err = Data0.LessSqrError(NN, Data_Out, u)+err;
               NN_BackPropagation Data1 = new
NN_BackPropagation(NN);  //
               Data1.UpDateWeightandThresh(Data_Out, u);  //
           }
           s++;
           }  //
           NN_W.NN_Weight_Modif_Back(NN, Chrom);
        }
        else
         //Algorithme GA seul
        {
           err=0;
           for(int u=1;u< (Nbr_Indiv+1); ++u)
            {
               Data0.UpDateInput(NN, Data_Out, u);
             for (int h=2; h<(Nbr_Layer+1);++h)
             {
                   Data0.UpDateLayer(NN,h);
             }
                 err = Data0.LessSqrError(NN, Data_Out, u)+err;
           }
           s++;
           NN_W.NN_Weight_Modif_Back(NN, Chrom);
        }
       err = 100* err / Nbr_Indiv;
       Fitness = 100-err;
       Indiv_New = true;
}

}
```

9.1.1.8 Indiv_Bin1_NN

```
package Ga.Individu;
import Tools.Gen_Aleat.Gen_Aleat;
import NN.Struct_In.*;
import NN.Struct_Out.*;
import NN.Struct.*;
import NN.FeedForward.*;
import NN.BackPropagation.*;
import NN.Pattern.*;
import Ga_Nn.Opt_Weight.*;
import Ga_Nn.Opt_Struct.*;
import java.io.*;


public class Indiv_Bin1_NN extends Indiv_Bin1
{
   private static int Nbr_Neur =6;
   private static int Nbr_In = 2;
   private static int Nbr_Out = 1;

   private static int iter_j = 1; //20
   private static int iter_BP = 10; //800
   private static String title = "Xor_v01";
        //String title = "Xor_v02";
      //String title = "Cancer_v01";
      //String title = "Cancer_v02";
      //String title = "Cancer_v03";
      //String title = "X1_Mult_X2_v01";
      //String title = "Symetrie_v01";

   private int Evaluation;
   public NN_Struct NN ;
   NN_Pattern_In Data_Out;
   NN_FeedForward Data0;

   double s = 0;
   double err=0;
   double err_tot = 0;

   NN_Struct_Modif NN1;
   NN_Initialisation NN_Init;

/************************************************/
public NN_Struct get_struct()
//  Renvoit la structure du réseau de neurones
{
      return NN;
}

/************************************************/
public void Set_Param(int New_Nbr_Neur, int New_Nbr_In, int
New_Nbr_Out, int New_iter_j, int New_iter_BP, String New_title)  {

   Nbr_Neur = New_Nbr_Neur;
   Nbr_In = New_Nbr_In;
   Nbr_Out = New_Nbr_Out;

   iter_j =  New_iter_j;
   iter_BP =  New_iter_BP;
```

```
   title =  New_title;

}

/*************************************************/
public Indiv_Bin1_NN()
// Constructeur
{
      try
      {
        String File_Data = "Samples/"  + title  + "/Pattern.txt";
        Data_Out = new NN_Pattern_In(File_Data);
        Data0 = new NN_FeedForward();
      }
    catch (IOException e)
      {
      }

  Nbr_Chrom = Nbr_Neur*Nbr_Neur;
  Nbr_Locus = 1;
  Nbr_Tot = Nbr_Chrom * Nbr_Locus;
  Chrom_Bin  = new int[Nbr_Tot];
  NN_Init = new NN_Initialisation(Chrom_Bin, Nbr_Neur, Nbr_In,
Nbr_Out);

}

/*************************************************/
public Object clonne()
//  Clone l'individu et le renvoit
{
      Indiv_Bin1_NN i = new Indiv_Bin1_NN();
      i.Nbr_Locus = this.Nbr_Locus;
    i.Nbr_Tot = this.Nbr_Locus;

      i.Nbr_Chrom = this.Nbr_Chrom;
      i.Nbr_In = this.Nbr_In;
      i.Nbr_Out = this.Nbr_Out;
      i.Nbr_Neur = this.Nbr_Neur;
      i.Indiv_New = this.Indiv_New;
      i.Fitness = this.Fitness;
      i.Fitness_Scaled = this.Fitness_Scaled;
      i.Fitness_Norma = this.Fitness_Norma;

      for(int j=0 ; j<Nbr_Chrom ; j++) {
            i.Chrom_Bin[j] =this.Chrom_Bin[j];
      }
      return i;
}

/*************************************************/
public void Indiv_Affich()
//  Affiche l'individu
{

  System.out.print("  ");
  int j=0;
  for(int i=0; i<Nbr_Chrom ; i++)
    { if(j==Nbr_Neur) {j=0; System.out.print(" ");}
      System.out.print("" + Chrom_Bin[i]  );
        ++j;
```

```
    }

    System.out.print("\tFit " + Fitness ) ;
    System.out.print("\tFitScl " + Fitness_Scaled ) ;
    System.out.println("\tFitNorm " + Fitness_Norma ) ;
}

/*************************************************/
public void Indiv_Eval()
// évaluation de l'individu
{
    NN_Struct_Modif NN_Modif = new NN_Struct_Modif( Chrom_Bin,
Nbr_Neur);
    NN = NN_Modif.Get_NN_New();
    NN.Set_Nbr_Intput(Nbr_In );
    NN.Set_Nbr_Output(Nbr_Out );
//       NN.Affich(0);
//       NN.Affich_Tab();

      int Nbr_Indiv = Data_Out.Get_Nbr_Sample();
      int Nbr_Layer = NN.Get_Nbr_Layer();
      if(Nbr_Layer>2){

      NN_Weight_Modif NN_W;
      int Nbr_Chrom = NN.Get_Nbr_Weight() + NN.Get_Nbr_Neur();
      double [] Chrom_Val = new double[Nbr_Chrom];

    for(int j=1; j < (iter_j+1) ; ++j)
    {
     for(int i=0; i<Nbr_Chrom ; i++)
       {  Chrom_Val[i] = g.Gen_Aleat_Double(-1 , 1 ) ; }
     NN_W = new NN_Weight_Modif(NN, Chrom_Val);
     s=0;

         for(int i=1; i<iter_BP ; ++i)
            {      err=0;
               for(int u=1;u< (Nbr_Indiv+1); ++u)
               {  Data0.UpDateInput(NN, Data_Out, u);
                    for (int h=2; h<(Nbr_Layer+1);++h) {
Data0.UpDateLayer(NN,h); }
                err = Data0.LessSqrError(NN, Data_Out, u)+err;
                  NN_BackPropagation Data1 = new
NN_BackPropagation(NN);
                  Data1.UpDateWeightandThresh(Data_Out, u);
               }
               s++;
               err = 100* err / Nbr_Indiv;
              }
              err_tot = err + err_tot;
         }
      err_tot = err_tot / iter_j ;
      Fitness =   100-err_tot ;
       }
       else Fitness = 0;
       Indiv_New = true;
}

}
```

9.1.2 Population

9.1.2.1 Pop_Abstract

```
package Ga.Population;

public interface Pop_Abstract {

     void ModifNombrePop(int i);
     void ModifTauxMut(double i);
     void ModifTauxCrois(double i);

     void Scaling(String d);
     void Simulated_Annealing_activation();
     void Affichage_Level(int i);

     void Elitisme_Set(double i);
     Population Initialisation();

     void Evaluation();
     void Simulated_Annealing();
     void Affiche();
     void Selection();
     void Croisement();
     void Mutation();

     double [] Affiche_crb();
}
```

9.1.2.2 Population

```
package Ga.Population;
import java.util.Date;
import Tools.Gen_Aleat.Gen_Aleat;
import Ga.Individu.*;
import Tools.Tri.*;
import java.util.*;

public abstract class Population implements Pop_Abstract {

   protected abstract Indiv_Abstract get_Indiv(int i);

   protected abstract Indiv_Abstract new_Individu();

   protected abstract Population New();
   protected abstract void CrossOver_Individus(Indiv_Abstract
Chr_P1,Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1,Indiv_Abstract
Chr_F2);
   protected abstract void Mutation_Individu(Indiv_Abstract  Chr_P,
Indiv_Abstract  Chr_F);

   Indiv_Abstract Data = new_Individu();

   Population next;
   Population prev;

   static protected int Affichage_Level;

   protected Gen_Aleat g = new Gen_Aleat();

   static protected int Gen_Num = 0;
```

```
   static private int Nbr_Indiv = 10;

   //Evolution
   static private boolean Croisement = true;
   static private double TauxCroisement = 0.60;
   static private boolean Mutation = true;
   static private double TauxMutation = 0.15;
   static private boolean Scaling = false;
   static private String Type_Scaling = "Scaling_Lin";
   static private boolean Elitisme = false;
   static private double TauxElitisme = 0.95;
   static protected boolean Simulated_Annealing = false;

   static protected Population Top;
   static protected Population Actif;
   static protected Population parcour;

   //Fitness
   static private double FitnessMax = Double.NEGATIVE_INFINITY;
   static private double FitnessMin = Double.POSITIVE_INFINITY;
   static private double FitSclMax = Double.NEGATIVE_INFINITY;
   static private double FitSclMin = Double.POSITIVE_INFINITY;
   static private double som;
   static private double som_scal;
   static private double som_norm;
   static private double moy;


   //Elitisme
   static protected int Elit;
   static protected int Nbr_Elit;

   //Simulated Annealing
   static private double T=0;
   static private double Ti=0, Tx=0, Tf=0;
   static private double DF_max=0;
   static private double DF_min=Double.POSITIVE_INFINITY;
   static private double CC1=0.9  ;
   static private double CC2=0.95  ;

   //Variable pour mesurere le temps
   static long now_old  = System.currentTimeMillis();

/************************************************/
private Population New(Population tail)
// Ajout d'un individu au début de la liste de la population
{
      Population l = New();
      l.prev = null;
      l.next = tail;
      if(tail!=null)    tail.prev = l;
      return(l);
}

/************************************************/
public Population Initialisation()
// Initialisation d'une population dont le nombre d'individus sera
"Nbr_Indiv"
{
  Population l = New(null);
```

```
  for(int i=0 ; i < Nbr_Indiv ; i++)
   {  l.Data = new_Individu();
      Population p = New(l);
      l=p;
    }
   Top = l;
   return(l);
}

/*************************************************/
// Méthode de Setup des paramètres

/*************************************************/
public void Scaling(String d)
// Activation de l'opération de scaling, des fitness après évaluation
// Le scaling est
// linéaire = "Lin",
// exponentiel = "Exp"
// ou de position = "Position"
{
   Scaling =true;
   Type_Scaling = d;
}

/*************************************************/
public void ModifTauxMut(double TauxMut)
// Active l'opération de mutation de la population avec un taux de
mutation égale a "TauxMut"
{
   TauxMutation = TauxMut;
   Mutation = true;
}

/*************************************************/
public void ModifTauxCrois (double TauxCrois)
// Active l'opération de croisment de la population avec un taux de
croisement égale à "TauxCrois"
{
   TauxCroisement = TauxCrois;
   Croisement = true;
}

/*************************************************/
public void ModifNombrePop(int N)
// Modifie le nombre d'individus de la population
{
   Nbr_Indiv = N;
}

/*************************************************/
public void Elitisme_Set (double d)
// Active l'opérateur d'élitisme lors de la sélection avec un taux
d'élitisme égale à d
{
   Elitisme = true;
   TauxElitisme = d;
}

/*************************************************/
public void Affichage_Level(int i)
```

```
// Modifie le niveau d'affichage utilisé lors de l'execution du
programme
// le nouveau taux d'affichage est de "i"
// "i" est compris entre 1 (affiche peu détaillé) et 6 (affichage
très détaillé)
{   Affichage_Level = i; }

/*************************************************/
void elit_in(int i)
//Modifie l'individu Elite de la population
{   Elit = i;  }

/*************************************************/
public void Simulated_Annealing_activation()
// Active l'opération de recuit simulé
{   Simulated_Annealing = true; }

/*************************************************/
/*************************************************/
// Méthode Get des paramètres
/*************************************************/
/*************************************************/
public int NombrePop()
// Renvoit le nombre d'individus
{ return Nbr_Indiv; }

/*************************************************/
public double TauxMut()
// Renvoit le taux de mutation
{ return TauxMutation ; }

/*************************************************/
public double TauxCrois()
// Renvoit le taux de croisement
{ return TauxCroisement; }

/*************************************************/
public boolean Scaling_actif_value()
// Renvoit true si le scaling est actif
{ return Scaling; }

/*************************************************/
public String Scaling_Type()
// Renvoit le type de scaling
{ return Type_Scaling; }

/*************************************************/
public double FitnessMax()
// Renvoit la Fitness Maximum
{ return FitnessMax; }

/*************************************************/
public double FitnessMin()
// Renvoit la Fitness Minimum
{ return FitnessMin; }

/*************************************************/
public double Somme()
// Renvoit la somme des Fitness
{ return som; }
```

```
/************************************************/
public Population Top()
// Renvoit le pointeur du sommet de la fille d'individu
{ return Top; }

/************************************************/
public double som_scal()
// Renvoit la somme des Fitness après mise à l'échelle
 { return  som_scal ; }

/************************************************/
public double som_norm()
// Renvoit la somme des Fitness après normalisation
{ return  som_norm ; }

/************************************************/
public double moy()
// Renvoit la moyenne des Fitness
{ return  moy ;  }

/************************************************/
/************************************************/
// Méthode d'Evaluation
/************************************************/
/************************************************/
public void Evaluation()
// Evaluation des fitness des individus, de la fitness Max, de la
fitness Min,
// et de l'individu Elite, de la moyenne moy, effectue un scaling sur
lesfitness si le
// Scaling est activé ensuite effectue la normalisation des fitness
entre 0 et 1
// et évalue les variable de recuit simulé si celui-ci est actif
{
        double g;
        FitnessMin = Double.POSITIVE_INFINITY;
        FitnessMax = Double.NEGATIVE_INFINITY;
        parcour = Top.next;
        int i = 0;
        som = 0;
        while (parcour!=null) {
             parcour.Data.Indiv_Eval();
              g = parcour.Data.Indiv_Fitness();
            som =  som + g;
            if( g > FitnessMax ) { FitnessMax = g; Elit=i;} ;
            if( g < FitnessMin ) { FitnessMin = g;} ;
            parcour=parcour.next;
            i++;
            }
        moy = som / Nbr_Indiv;

    if(Scaling==true)
     {
       if(Type_Scaling == "Lin")  { Scaling_Lin(0); }
       if(Type_Scaling == "Exp")  { Scaling_Exp(Gen_Num, 1000000); }
            if(Type_Scaling == "Position")  { Scaling_Posi(); }
     };
    Normalisation();

    if(Simulated_Annealing==true) {
        if(Gen_Num==0) {   Simulated_Annealing_Init();  }
```

```
      else {   Simulated_Annealing(); };
    }
}

/************************************************/
/************************************************/
// Méthode de Normalisation
/************************************************/
/************************************************/
void Normalisation()
// Entre 0 et 1 normalisation des Fitness ou des Fitness après
Scaling
{
   double a,b, g, min_n, max_n;

   parcour = Top.next;
   som_norm =0;
   min_n = 0.0;
   max_n = 1.0;
   int i = 0;

   if (Scaling == true)
   {
      if (FitSclMax!=FitSclMin)
       {
            a = (max_n - min_n) / (FitSclMax - FitSclMin);
            b = ( (min_n * FitSclMax) - (FitSclMin * max_n) ) /
(FitSclMax - FitSclMin);

            if(Affichage_Level>=6) {   System.out.println(" a : " +
a + "\t b : " + b ) ; } ;
            while (parcour!=null)
            {
                g = parcour.Data.Indiv_Fitness_Scl();
                g = g*a +b;
                som_norm =   som_norm + g;
                parcour.Data.Indiv_in_Fit_Norma(g);
                parcour=parcour.next;
                i++;
            }
       }
   }
   else
   {
      if (FitnessMax!=FitnessMin)
      {
        a = (max_n - min_n) / (FitnessMax - FitnessMin);
        b = (min_n * FitnessMax - FitnessMin * max_n) / (FitnessMax -
FitnessMin);
        if(Affichage_Level>=6) {   System.out.println(" a : " + a +
"\t b : " + b ) ; } ;
        while (parcour!=null)
        {
           g = parcour.Data.Indiv_Fitness();
           g = g*a +b;
           som_norm =   som_norm + g;
           parcour.Data.Indiv_in_Fit_Norma(g);
           parcour=parcour.next;
           i++;
        }
     }
```

```
  }
}

/************************************************/
/************************************************/
// Méthode de Scaling
/************************************************/
/************************************************/
void Scaling_Lin(int d)
// Scaling Linéaire entre d et (d+1)
{
   double a,b, g, min_n, max_n;

   parcour = Top.next;
   som_scal =0;

   if(d>10) { d=10; };
   if(d<0) { d=0; };

   min_n = 0.0 + d;
   max_n = 1.0 + d;

   if (FitnessMax!=FitnessMin) {
     a = (max_n - min_n) / (FitnessMax - FitnessMin);
     b = (min_n * FitnessMax - FitnessMin * max_n) / (FitnessMax -
FitnessMin);

     if(Affichage_Level>=6) {   System.out.println(" a : " + a + "\t
b : " + b ) ; } ;
     while (parcour!=null) {
          g = parcour.Data.Indiv_Fitness();
          g = g*a +b;
              if( g > FitSclMax) { FitSclMax= g;} ;
              if( g < FitSclMin) { FitSclMin= g;} ;
          som_scal =   som_scal + g;
          parcour.Data.Indiv_in_Fit_Scl(g);
          parcour=parcour.next;
        }
   }
}

/************************************************/
void Scaling_Posi()
// Scaling en fonction de la position, les individus sont classés en
fonction de la valeur de leur fitness
{
   double g;

   parcour = Top.next;
   som_scal =0;
   double[] tab = new double[Nbr_Indiv];

   int i =0;
   while (parcour!=null)
      {   g = parcour.Data.Indiv_Fitness();
          som_scal = som_scal + g;
          tab[i] = g;
          parcour=parcour.next;
            ++i;
      }
      --i;
```

```
      FitSclMax = i;
        FitSclMin =0;

      Tri  T = new Tri(tab);

      LinkedList l = T.Get_l();
      int[] tab3 =  T.Get_resu();
    parcour = Top.next;
    i=0;

    while (parcour!=null)
    {     int d5 = tab3[i];
          parcour.Data.Indiv_in_Fit_Scl(d5);
          parcour=parcour.next;
          ++i;
    }
}

/************************************************/
   void Scaling_Exp(int N, int N_max) {
/************************************************/

   double a,b, g;
   double k, s;
   double p = 0.1;

   parcour = Top.next;

   k = Math.tan ( ( 3.14159 / 2.0 * N / ( N_max + 1 ) ) ) ;
   s = Math.pow(k,p) ;
   if(Affichage_Level>=6) {   System.out.println(" k : " + k + "\t s
: " + s ) ; } ;

   while (parcour!=null) {
      g = parcour.Data.Indiv_Fitness();
      g = Math.pow(g,s);
      parcour.Data.Indiv_in_Fit_Scl(g);
      parcour=parcour.next;
     }
}

/************************************************/
/************************************************/
// Methode de Croisement
/************************************************/
/************************************************/
public void Croisement()
// Croisement d'individus
{
   double m, n;
   int nc = (int)Math.floor( TauxCroisement * Nbr_Indiv/(2.0) );
   if(Affichage_Level>=3) {  System.out.println("Croisement :"); };
   for(int i=0;i<nc;i++)
   {
      m = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Indiv) );
      n = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Indiv) );
        while( n== Elit )
        {
           n = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Indiv) );
        };
        while( m== Elit || m==n)
```

```
        {
           m = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Indiv) );
        };

      if(Affichage_Level>=3)
      {
         System.out.print(" m : " + (int)m );
         System.out.println(" n : " + (int)n );
      }

      int mm = (int)m;
      int nn =  (int)n ;
      double Ef, Ei;
      int r;

      Indiv_Abstract Chr_F1 = new_Individu();
      Indiv_Abstract Chr_F2 = new_Individu();

        Indiv_Abstract Chr_P1 = get_Indiv(mm);
      Indiv_Abstract Chr_P2 = get_Indiv(nn);

      if(Affichage_Level>=4) {  Chr_P1.Indiv_Affich(); } ;
      if(Affichage_Level>=4) {  Chr_P2.Indiv_Affich(); };

      CrossOver_Individus(Chr_P1,Chr_P2, Chr_F1,Chr_F2);

       // Simulted annealing
       if(Simulated_Annealing ==true)
       {  Chr_F1.Indiv_Eval();
          Chr_F2.Indiv_Eval();

          Ef = Chr_F1.Indiv_Fitness();  //fils
          Ei = Chr_P1.Indiv_Fitness();  //père
          r= Simulated_Annealing_pT(Ei, Ef);
          if(r==2) { remplacer(mm, Chr_F1); };

          Ef = Chr_F2.Indiv_Fitness();  //fils
          Ei = Chr_P2.Indiv_Fitness();  //père
          r= Simulated_Annealing_pT(Ei, Ef);
          if(r==2) { remplacer(nn, Chr_F2); };
       }
       else
       {  Chr_F1.Set_Indiv_New(false);  // false car évaluation pas
faite
              remplacer(mm, Chr_F1);
          Chr_F2.Set_Indiv_New(false);  // false car évaluation pas
faite
          remplacer(nn, Chr_F2);
       }

      if(Affichage_Level>=4) { Chr_F1.Indiv_Affich(); } ;
      if(Affichage_Level>=4) { Chr_F2.Indiv_Affich(); } ;

    }
    if(Affichage_Level>=4) { System.out.println("");  Affiche_Pop();
} ;
}

/*************************************************/
/*************************************************/
// Methode de Mutation
```

```
/*************************************************/
/*************************************************/
public void Mutation()
// Mutation d'individu
{
double d, j;
double Ei, Ef;

int n, nm, r;
nm = (int)Math.floor( TauxMutation * Nbr_Indiv );

   if(Affichage_Level>=3) {  System.out.println("Mutation :"); };
   for(int i=0;i<nm;i++)
   {
      n = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Indiv) );
      while( n== Elit )
        {  n = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Indiv) );
        };
      if(Affichage_Level>=3) {    System.out.println(" mut : " + n );
};

      Indiv_Abstract  Chr_F = new_Individu();
      Indiv_Abstract Chr_P = get_Indiv(n);

      if(Affichage_Level>=4) { Chr_P.Indiv_Affich(); } ;

      Mutation_Individu(Chr_P, Chr_F);

      if(Affichage_Level>=4) {  Chr_F.Indiv_Affich(); };

      if (Simulated_Annealing == true)
      {   Chr_F.Indiv_Eval();
          Ef = Chr_F.Indiv_Fitness();  //fils
          Ei = Chr_P.Indiv_Fitness();  //père
          r= Simulated_Annealing_pT(Ei, Ef);
          if(Affichage_Level>=4) {
System.out.println("Simulated_Annealing r = " + r); } ;
          if(r==2) { remplacer(n, Chr_F); };
       }
       else
       {
              Chr_F.Set_Indiv_New(false);  // false car évaluation
pas faite
              remplacer(n, Chr_F);
       }
   }
   if(Affichage_Level>=4) {System.out.println(""); Affiche_Pop(); } ;
}

/*************************************************/
/*************************************************/
// Méthode de Selection
/*************************************************/
/*************************************************/
public void Selection()
//Sélection d'une nouvelle population et update du numéro de
génération
{

    // Selection de la roulette
    Roulette_Whell();
```

```
    if(Affichage_Level>=3) { System.out.println("Après Roulettwheel :
") ; Affiche_Pop(); } ;

    Gen_Num++;

}

/*************************************************/
public void Roulette_Whell()
/*************************************************/
// Sélection avec la méthode de la roulette Whell
{
   Population Pop2  = New(null);

   if(Elitisme == true) {
    // ajout des elites à la population de Pop2
       Pop2 = Elitisme(Pop2);
     }

    if(Affichage_Level>=3) {  System.out.println("Selection : "); };

   double c, delta_old, delta;
   double som_norma = som_norm();

   if(Affichage_Level>=3) {  System.out.print(" ["); };
   for(int k=0 ; k < (Nbr_Indiv-Nbr_Elit) ; k++)
        {
                int jj;
         delta = g.Gen_Aleat_Double(0, som_norma);
         jj =0;
           parcour = Top.next;
           while (parcour!=null)
             {
                 if(delta>=0)
                   {
                     jj++;
                 c = parcour.Data.Indiv_Fitness_Norma();
                     delta_old = delta - c;
                     if(Affichage_Level>=5) {
System.out.println("delta : " + delta + "\tsub : " + c + "\t
delta_old : " + delta_old + " jj : " + jj ); } ;
                     delta = delta_old;
                 parcour = parcour.next;

                     if(c<=0 && parcour==null && jj>0)
                      {
                          boolean flag1 = false;
                          while (flag1 == false)
                            {
                              jj--;
                              c =
get_Indiv(jj).Indiv_Fitness_Norma();
                              if (c!=0)
                                 {
                                    flag1 =true;
                                    if(Affichage_Level>=5) {
System.out.print(" " + jj); }
                                 }
                            }
                      }
```

```
                    }
                  else
                    {
                         parcour =null ;
                  } ;
              }
              Pop2.Data = get_Indiv(jj-1);
              Population 1 = New(Pop2);
              Pop2 = 1;
              if(Affichage_Level>=3) { System.out.print(" " + (jj-1) );
} ;
         }

     if(Affichage_Level>=3) {  System.out.println(" ]"); };

     Top = Pop2;
}

/************************************************/
/************************************************/
// Méthode d' Elitisme
/************************************************/
/************************************************/
private Population Elitisme(Population Pop2)
// Sélectionne les élites et les places dans la nouvelle population
lors de l'étape de sélection
{
     int j=0;
     int w=0;
     double cc;
     double c_max=Double.NEGATIVE_INFINITY;


 // z est utilisé pour plafonner le nombre d'elites sinon on sature
la population
     int z = (int)Math.floor ( Nbr_Indiv * (1.0-TauxElitisme) );
     if(Affichage_Level>=3) {  System.out.println("z :" + z); };

 // z mais il faut garder au moins une elite
     if (z<1) { z=1; };

     if(Affichage_Level>=3) {  System.out.println("Elitisme :"); };
     if(Affichage_Level>=3) {  System.out.print(" ["); };


              Pop2.Data = get_Indiv(Elit);
              Population L = New(Pop2);
              Pop2 = L;
              c_max = Pop2.Data.Indiv_Fitness_Norma();

     if(Affichage_Level>=3) {  System.out.print(" " + Elit ); } ;

     int y =0;
     int y_max =0;
     parcour = Top.next;
     while (parcour!=null && y<(z-1) )
        {
           cc = parcour.Data.Indiv_Fitness_Norma();
           if ( (cc>=TauxElitisme) && (j!=Elit) )
             {
```

```
                    Pop2.Data = get_Indiv(j);
                     L = New(Pop2);
                    Pop2 = L;
                  if(Affichage_Level>=3) {  System.out.print(" " + j );
} ;
                  if(cc>c_max) { w = j;  c_max = cc ; y_max = y;   };
                  y++;
              };
            j++;
            parcour = parcour.next;
        }

    if(Affichage_Level>=3) { System.out.println(" ]"); } ;

    Pop2.elit_in(Nbr_Indiv-1);

    Nbr_Elit=y+1;

    return Pop2;     //new
}


/*************************************************/
/*************************************************/
// Methode de Simulated_Annealing
/*************************************************/
/*************************************************/
private void Simulated_Annealing_Init()
// Methode de recuit simulé initialisant les variables
{

Population parcour1 = New(null);
Population parcour2 = New(null);
double c1, c2, r;
int N1, N2, N;

parcour1 = Top.next;
int j = 0;
int i_max=0;
int j_max=0;
int i_min=0;
int j_min=0;

while (parcour1!=null) {

if(parcour1.next!=null) {
    parcour2 = parcour1.next;
    int i = 0;
    while (parcour2!=null) {
         c1 = parcour1.Data.Indiv_Fitness();
         c2 = parcour2.Data.Indiv_Fitness();
         if (c2>c1) {r=c2-c1;}
         else { r=c1-c2; };
         if (r>DF_max) {
            DF_max=r;
            i_max =i;
            j_max =j;
            };
         if (r<DF_min) {
            DF_min=r;
            i_min =i;
```

```
            j_min =j;
                };
          i++;
         parcour2 = parcour2.next;
         }
        }
       parcour1 = parcour1.next;
       j++;

      }
        Ti=-DF_max / (Math.log( (1.0/ 0.75)-1) ) ;
        Tx=-DF_max / (Math.log( (1.0/ 0.99)-1) ) ;
        Tf=-DF_min / (Math.log( (1.0/ 0.99)-1) ) ;
        System.out.println("Ti : " + Ti + " Tx : " + Tx + " Tf : " +
Tf);
        N1=(int)Math.floor( (Math.log( Tx/ Ti ) ) / (Math.log(CC1) ) );
        N2=(int)Math.floor( (Math.log( Tf/ Tx ) ) / (Math.log(CC2) ) );
        N = N1+N2;
        System.out.println("N1 : " + N1 + " N2 : " + N2 + " N : " + N
);

      T=Ti;
}

/*************************************************/
public void Simulated_Annealing()
// Méthode de recuit simulé update la varialbe T du recuit simulé
{
   if(T>Tx) { T=CC1*T; }
      else { T=CC2*T; }
//   System.out.println("T : " + T) ;
}

/*************************************************/
protected int Simulated_Annealing_pT(double Ei, double Ef)
// Ei père et Ef Fils
/// Si on garde le père on retourne la valeur 1
/// Si on garde le fils on retourne la valeur 2
{

   double pT, prob ;
   pT = (1 / ( 1 + Math.exp( -(Ei-Ef)/T ) ) );
   // pT =1/2 si exp = 1 => si T>> ou Ei=Ef => garde fils
   // pT =1 si exp = 0 => si T<< ou Ei >> Ef   => garde père
   prob = g.Gen_Aleat_Double(0, 1);
   if (prob<pT) { return 1;}
   // garde père
   else {return 2;}
   // garde fils
}

/*************************************************/
/*************************************************/
// Méthode Affichage
/*************************************************/
/*************************************************/
public void Affiche()
// Méthode d'affichage
{
   if(Affichage_Level>=2) {
//    System.out.println("");
```

```
      System.out.println("generation : " + Gen_Num );
      Affiche_Pop();

      System.out.print("Elit : " +    Elit );
      System.out.print("\tFitmax : " +   FitnessMax() );
      System.out.println("\tMoyenne : " + moy() );
   }
   else {

   long now = System.currentTimeMillis();
   now = (now - now_old)/1000;
   System.out.print("Gen_Num :   " + Gen_Num );
   System.out.print("  Time :   " + now);
   System.out.print("\t FitnessMax" +  FitnessMax() );
   System.out.println("\t moy " + moy() );
   }
}

/*************************************************/
public void Affiche_Pop() {

   parcour = Top.next;
   int s=0;
   while (parcour!=null) {
           System.out.print(" " + s++);
           parcour.Data.Indiv_Affich();
           parcour=parcour.next;
           }
}

/*************************************************/
public double [] Affiche_crb()
// Dessin de la courbe return
{
   double[] valu;
   valu = new double[3];
   valu[0]= FitnessMax();
   valu[1]= moy();
   valu[2]= FitnessMin();
   return valu;
}

/*************************************************/
public void remplacer(int u, Indiv_Abstract t) {
       Population cell1 = New(null);
       cell1.Data = t;

      parcour = Top.next;
      if(u==0) { cell1.next = parcour.next ; cell1.prev=null;
Top.next=cell1; }
      else {
      for(int i=0;i<u;i++) {
         parcour=parcour.next;
        };
         cell1.next = parcour.next;
         cell1.prev = parcour.prev;
         if (parcour.prev!=null) {parcour.prev.next = cell1; } ;
         if (parcour.next!=null) {parcour.next.prev = cell1; } ;
       }
}
}
```

9.1.2.3 Popu1

```
package Ga.Population;
import Ga.Individu.*;

public class Popu1 extends Population {

protected Population New()
// Renvoit une classe Popu1 sous classe de Population
{
   Popu1 s1 = new Popu1();
   return s1;
}

/*************************************************/
protected void CrossOver_Individus(Indiv_Abstract
Chr_P1,Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1,Indiv_Abstract
Chr_F2)
// Méthode de croisement "Floating_CrossOver"
{  double alpha, j, k, d1, d2;
      int Nbr_Chrom;
      Double ObjDbl;

      Nbr_Chrom = Chr_P1.Nbr_Chrom_Get() ;
      for(int i=0 ; i<Nbr_Chrom ; i++)
      {
            alpha = g.Gen_Aleat_Double(-0.5 , 1.5) ;
       ObjDbl = (Double)Chr_P1.Chromo_Val(i);
            j = ObjDbl.doubleValue();
       ObjDbl = (Double)Chr_P2.Chromo_Val(i);
         k = ObjDbl.doubleValue();

         d1 = ( (1.0-alpha) * j ) + (alpha * k);
         ObjDbl = new Double(d1);
         Chr_F1.Chromo_in(i, ObjDbl);

         d2 =  ( (1.0-alpha) * k ) + (alpha * j);
         ObjDbl = new Double(d2);
         Chr_F2.Chromo_in(i, ObjDbl);
       };
}

/*************************************************/
protected void Mutation_Individu(Indiv_Abstract  Chr_P,
Indiv_Abstract  Chr_F)
// Méthode de mutation "Floating_Mutation"
{
   double j;

       int Nbr_Chrom = Chr_P.Nbr_Chrom_Get();
       int w = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Chrom) );
       for(int u=0 ; u<Nbr_Chrom ; u++)
       {
         if(w!=u)
         {
              Double ObjDbl = (Double)Chr_P.Chromo_Val(u);
              j = ObjDbl.doubleValue();
         }
         else
         {
              Double ObjDbl = (Double)Chr_P.Chromo_Gen_Aleat(u);
```

```
            j = ObjDbl.doubleValue();
          }
          Double ObjDbl = new Double(j);
          Chr_F.Chromo_in(u, ObjDbl);
        }
}

/*************************************************/
protected Indiv_Abstract get_Indiv(int i)
// Renvoit l'individu i
{
       parcour = Top.next;
     int j=0;
     while (parcour!=null) {
        if (i==j) {
            Indiv1 u = (Indiv1)parcour.Data.clonne();
            return u;
            };
        j++;
        parcour=parcour.next;
      }
     return null;
}

/*************************************************/
public Indiv_Abstract new_Individu()
// Crée un nouvel individu
{
    Indiv1 ind = new Indiv1();
    return ind;
}

}
```

9.1.2.4 Popu_Bin1

```
package Ga.Population;
import Ga.Individu.*;

public class Popu_Bin1 extends Population {

protected Population New()
// Renvoit une classe Popu_Bin1 sous classe de Population
{
   Popu1 s1 = new Popu1();
   return s1;
}

/*************************************************/
protected void CrossOver_Individus(Indiv_Abstract
Chr_P1,Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1,Indiv_Abstract
Chr_F2)
// Méthode de croisement de chaine binaire à 1 point ou à 2 points
{  double alpha1, alpha2, temp;
   Integer ObjInt;
   String Type_Crossover;

   int Nbr_Chrom = Chr_P1.Nbr_Chrom_Get() ;
   int Nbr_Locus = Chr_P1.Nbr_Locus_Get();
   int Nbr_Tot =  Nbr_Chrom + Nbr_Locus -1;
```

```
   //Type_Crossover = "1_Point";
   Type_Crossover = "2_Points";
   if( Type_Crossover == "2_Point" )  alpha1 = g.Gen_Aleat_Double(0 ,
Nbr_Tot) ;
   else alpha1 = 0;

   alpha2 = g.Gen_Aleat_Double(0 , Nbr_Tot) ;

   if( alpha1 > alpha2)
   {  temp = alpha1;
      alpha1 =  alpha2;
      alpha2 = temp;
   }
      for(int i=0 ; i<Nbr_Tot ; i++)
      {
          ObjInt = (Integer)Chr_P1.Chromo_Val(i);
          if(alpha1<=i && i<=alpha2) { Chr_F1.Chromo_in(i, ObjInt); }
          else { Chr_F2.Chromo_in(i, ObjInt); }

          ObjInt = (Integer)Chr_P2.Chromo_Val(i);
          if(alpha1<=i && i<=alpha2) { Chr_F2.Chromo_in(i, ObjInt); }
          else { Chr_F1.Chromo_in(i, ObjInt); }
      };
}

/*************************************************/
protected void Mutation_Individu(Indiv_Abstract  Chr_P,
Indiv_Abstract  Chr_F)
// Méthode de mutation de chaine binaire
{
    double j;
    Integer ObjInt;

    int Nbr_Chrom = Chr_P.Nbr_Chrom_Get() ;
    int Nbr_Locus = Chr_P.Nbr_Locus_Get();
    int Nbr_Tot =  Nbr_Chrom + Nbr_Locus -1;

    int w = (int)Math.floor(g.Gen_Aleat_Double(0,Nbr_Tot) );
    for(int i=0 ; i<Nbr_Tot ; i++)
      {  if(w!=i) ObjInt = (Integer)Chr_P.Chromo_Val(i);
         else
         {
              ObjInt = (Integer)Chr_P.Chromo_Val(i);
              j = ObjInt.intValue();
              if(j==0) ObjInt = new Integer(1);
           else ObjInt = new Integer(0);
         }
         Chr_F.Chromo_in(i, ObjInt);
      }
}

/*************************************************/
protected Indiv_Abstract get_Indiv(int i)
// Renvoit l'individu i
{
     parcour = Top.next;
     int j=0;
     while (parcour!=null) {
        if (i==j) {
            Indiv_Bin1 u = (Indiv_Bin1)parcour.Data.clonne();
            return u;
```

```
            };
          j++;
          parcour=parcour.next;
        }
      return null;
}

/*************************************************/
public Indiv_Abstract new_Individu()
// Crée un nouvel individu
{
    Indiv_Bin1 ind = new Indiv_Bin1();
    return ind;
}

}
```

9.1.2.5 Popu1_Ga_NN_BP_Param

```
package Ga.Population;
import Ga.Individu.*;
import NN.Struct.*;
import java.io.*;

public class Popu1_Ga_NN_BP_Param extends Popu1 {

   private static String title = "Xor_v02";
   private static int iter_BP = 20;
   private static int iter_j = 50; //50

/*************************************************/
public void Set_Param(int New_iter_j , int New_iter_BP, String
New_title)  {

   iter_BP =  New_iter_BP;
   title =  New_title;
   iter_j =  New_iter_j ;

}

/*************************************************/
protected Indiv_Abstract get_Indiv(int i)
// Renvoit l'individu i
{
     parcour = Top.next;
     int j=0;
     while (parcour!=null) {
        if (i==j) {
            Indiv1_BP_Param u =
(Indiv1_BP_Param)parcour.Data.clonne();
            return u;
            };
        j++;
        parcour=parcour.next;
      }
     return null;
}

/*************************************************/
public Indiv_Abstract new_Individu()
// Crée un nouvel individu
```

```
{
    Indiv1_BP_Param ind = new Indiv1_BP_Param();
    ind.Set_Param(iter_j , iter_BP, title);
    return ind;
}

/************************************************/
public void Affiche()
// Affiche les individus de la population
{
   if(Affichage_Level>=2) {
      System.out.println("");
      System.out.println("generation : " + Gen_Num );
      Affiche_Pop();

      System.out.print("Elit : " +    Elit );
      System.out.print("\tFitmax : " +   FitnessMax() );
      System.out.println("\tMoyenne : " + moy() );
   }
   else {

   long now = System.currentTimeMillis();
   now = (now - now_old)/1000;
   System.out.print("Gen_Num:  " + Gen_Num );
   System.out.print(" time:  " + now);
   System.out.print("\t fitness max:  " +  ( FitnessMax() ) );
   System.out.print("\t  moy:  " + moy() );
   Indiv_Abstract  Chr4 = get_Indiv(Elit);
   Chr4.Indiv_Affich();

   }
}

}
```

9.1.2.6 Popu1_Ga_NN_Davis

```
package Ga.Population;
import Ga.Individu.*;
import NN.Struct.*;
import java.io.*;

public class Popu1_Ga_NN_Davis extends Popu1 {

   private static boolean flag_BP = false;
   private static String title = "Xor_v02";
   private static int iter_BP = 100;

/************************************************/
public void Set_Param(boolean New_flag_BP, int New_iter_BP, String
New_title)  {

   iter_BP =  New_iter_BP;
   title =  New_title;
   flag_BP =  New_flag_BP;

}


/************************************************/
protected Indiv_Abstract get_Indiv(int i) {
```

```
     parcour = Top.next;
     int j=0;
     while (parcour!=null) {
        if (i==j) {
            Indiv1_NN u = (Indiv1_NN)parcour.Data.clonne();
            return u;
            };
        j++;
        parcour=parcour.next;
      }
     return null;
}

/*************************************************/
public Indiv_Abstract new_Individu(){
    Indiv1_NN ind = new Indiv1_NN();
    ind.Set_Param(flag_BP, iter_BP, title);
    return ind;
}

/*************************************************/
public void Affiche() {
   if(Affichage_Level>=2) {
      System.out.println("");
      System.out.println("generation : " + Gen_Num );
      Affiche_Pop();

      System.out.print("Elit : " +    Elit );
      System.out.print("\tFitmax : " +   FitnessMax() );
      System.out.println("\tMoyenne : " + moy() );
   }
   else {

   long now = System.currentTimeMillis();
   now = (now - now_old)/1000;
   System.out.print("   " + Gen_Num );
   System.out.print("   " + now);
   System.out.print("\t" +  (100 - FitnessMax() ) );
   System.out.println("\t" + moy() );
//   get_Indiv(Elit).Indiv_Affich();

   }
}

/*************************************************/
protected void CrossOver_Individus_(Indiv_Abstract
Chr_P1,Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1,Indiv_Abstract
Chr_F2)
{  double alpha, temp;
   int u1, u2,u3;

   boolean flag = true;
   if(flag ==true)
   { super.CrossOver_Individus(Chr_P1, Chr_P2, Chr_F1, Chr_F2);
   }
   else
   {    Indiv1_BP_Param  Chr3 =  (Indiv1_BP_Param)Chr_P1;
        NN_Struct NN = Chr3.NN;
        int Nbr_Neur = NN.Get_Nbr_Neur();

        for(int k=0; k<2; ++k)
```

```
        {
            alpha = g.Gen_Aleat_Double(0, 1) ;
            alpha = (Nbr_Neur+1)*alpha;
            int n = (int) alpha;
            //avec n entre 1 et 6 pour XOr_v01
          if(Affichage_Level>=4) {  System.out.print("crs : " + n + "
"); }

            u1= NN.Get_VP_i(n);
            u3= NN.Get_VP_i(n+1);
            u3=u3-u1;
          u2= u1-1+n-1;
            for(int j=(u2); j<(u2+u3+1); ++j)
            {   Double ObjDbl = (Double)Chr_F2.Chromo_Val(j);
                double k1 = ObjDbl.doubleValue();
                ObjDbl = (Double)Chr_F1.Chromo_Val(j);
                double k2 = ObjDbl.doubleValue();
                temp = k1;
                k1 = k2;
                k2 = temp;
                ObjDbl = new Double(k1);
                Chr_F2.Chromo_in(j, ObjDbl);
                ObjDbl = new Double(k2);
                Chr_F1.Chromo_in(j, ObjDbl);
            }
        }
    }
}

}
```

9.1.2.7 Popu_Bin1_Struct2

```
package Ga.Population;
import Ga.Individu.*;
import Ga_Nn.Opt_Struct.*;
import NN.Struct.*;

//public class Popu_Bin1_Struct extends Population {
public class Popu_Bin1_Struct2 extends Popu_Bin1 {

protected Population New()
// Renvoit une classe Popu_Bin1 sous classe de Population
{
   Popu_Bin1_Struct2 s1 = new Popu_Bin1_Struct2();
   return s1;
}

/************************************************/
protected void CrossOver_Individus_(Indiv_Abstract
Chr_P1,Indiv_Abstract Chr_P2, Indiv_Abstract Chr_F1,Indiv_Abstract
Chr_F2)
// Méthode de croisement
{   System.out.println("-1 ") ;
   Indiv_Bin1_NN  Chr3 =  (Indiv_Bin1_NN)Chr_P1;
    System.out.println("0 ") ;
        NN_Struct NN = Chr3.NN;
    System.out.println("0 ") ;
        int Nbr_Neur = NN.Get_Nbr_Neur();

//    Chr_F1 =  (Indiv_Bin1_NN)Chr_P1.clonne();
```

```
    System.out.println("0 ") ;
    Indiv_Bin1_NN P1 =  (Indiv_Bin1_NN) Chr_F1;
    System.out.println("1 ") ;

    Chr_F2 =  (Indiv_Bin1_NN)Chr_P2.clonne();
    Indiv_Bin1_NN P2 =  (Indiv_Bin1_NN) Chr_F2;
    System.out.println("2 ") ;

//    int Nbr_Neur =  P1.NN.Get_Nbr_Neur();
    int  alpha2 = (int) g.Gen_Aleat_Double(0 , Nbr_Neur);

   NN_CrossOver cros = new NN_CrossOver(P1.Chrom_Bin, P2.Chrom_Bin,
alpha2, Nbr_Neur);
}

/*************************************************/
//protected void Mutation_Individu(Indiv_Bin1_NN  Chr_P,
Indiv_Abstract  Chr_F)
protected void Mutation_Individu(Indiv_Bin1_NN  Chr_P, Indiv_Bin1_NN
Chr_F)
// Méthode de mutation des connexions
{
    Chr_F =  (Indiv_Bin1_NN)Chr_P.clonne();

    int Nbr_Neur =  Chr_P.NN.Get_Nbr_Neur();
    int Nbr_In  = Chr_P.NN.Nbr_Input();
    int Nbr_Out = Chr_P.NN.Nbr_Output();

    int  alpha2 = (int) g.Gen_Aleat_Double(0 , (Nbr_Neur+1));
    NN_Mutation mut = new NN_Mutation( Chr_F.Chrom_Bin , alpha2 ,
Nbr_In, Nbr_Out);
}

/*************************************************/
protected Indiv_Abstract get_Indiv(int i)
// Renvoit l'individu i
{
     parcour = Top.next;
     int j=0;
     while (parcour!=null) {
        if (i==j) {
            Indiv_Bin1_NN u = (Indiv_Bin1_NN)parcour.Data.clonne();
            return u;
            };
        j++;
        parcour=parcour.next;
      }
     return null;
}

/*************************************************/
public Indiv_Abstract new_Individu()
// Crée un nouvel individu
{
    Indiv_Bin1_NN ind = new Indiv_Bin1_NN();
    return ind;
}

}
```

9.1.3 NN

9.1.3.1 Struct

```
package NN.Struct;

import java.lang.Math;
import java.util.Vector;
import Tools.Gen_Aleat.Gen_Aleat;

/*********************************************************************/
public class NN_Struct
{
   Gen_Aleat Alea = new Gen_Aleat();
   double al = Alea.Gen_Aleat_Double() ;  // Variable généré
aléatoirement

   Vector VC = new Vector();  // Vecteur des couches
   Vector VP = new Vector();  // Vecteur des neurones précédents VP
   Vector VS = new Vector();  // Vecteur des neurones précédents VS
   Vector Weight = new Vector();  // Vecteur des poids des connexions
   Vector Thresh = new Vector();  // Vecteur des offsets des neurones
   Vector State = new Vector();   // Vecteur des états des neurones
   Vector Actfn = new Vector();   // Vecteur des activations des
neurones
   Vector VP_To = new Vector();   // Vecteur des neurones suivant VP
   Vector VS_To = new Vector();   // Vecteur des neurones suivant VS
   Vector Link = new Vector();   // Vecteur de liens entre VP et
VP_To
   int Nbr_Neur;  // Nombre de neurones
   int Nbr_Neur_Courant;  // Nombre de neurones courant lors de la
construction
   int Nbr_Poids;  // Nombre de poids
   int Nbr_Input;  // Nombre d'entrées du réseau
   int Nbr_Output;   // Nombre de sorties du réseau

   int Act_In[];   // Vecteur d'activation des neurones d'entrée

/*********************************************************************/
   public NN_Struct(int Nbr_N)
   // Constructeur
   {
      Nbr_Neur=Nbr_N;
      Nbr_Neur_Courant=1;
    int Nbr_Neur_Courant =0;
      VC.addElement(new Integer(0));
      VS.addElement(new Integer(0));
      VP.addElement(new Integer(Nbr_Neur + 1));
      VS_To.addElement(new Integer(0));
      VP_To.addElement(new Integer(Nbr_Neur + 1));

      for(int i=1; i < (Nbr_Neur + 2); ++i)
        {
          VP.addElement(new Integer(1));
          VP_To.addElement(new Integer(1));
        }
   }

/*********************************************************************/
   public void New_Act_In()
   //Création des vecteurs d'activation pour les entrées
```

```
    {
       Act_In = new int[(Nbr_Input+1)];
       Act_In[0]=Nbr_Input;
       for(int i=1; i<(Nbr_Input+1); ++i)
       {  Act_In[i]=1; }
    }

/*********************************************************************/
    public void Affich_Act_In()
    //Afficher le vecteur d'activation pour les neurones d'entrée
    {
       System.out.println("Nbr_Input: " + Nbr_Input + " ");
       for(int i=1; i<(Nbr_Input+1); ++i)
       {  System.out.print("" + Act_In[i] ); }
    }

/*********************************************************************/
    public void insert_i_Thresh(int i, int u)
    //inserrer un entier u dans le vecteur des offsets à la position
i
    {
       Thresh.insertElementAt(new Integer(u), i);
    }

/*********************************************************************/
    public void insert_i_Thresh(int i, double u)
    //inserrer un réel u dans le vecteur des offsets à la position  i
    {
       Thresh.insertElementAt(new Double(u), i);
    }

/*********************************************************************/
    public void insert_i_State(int i, int u)
    //inserrer un entier u dans le vecteur des états à la position  i
    {
       State.insertElementAt(new Integer(u), i);
    }

/*********************************************************************/
    public void insert_i_State(int i, double u)
    //inserrer un réel u dans le vecteur des états à la position  i
    {
       State.insertElementAt(new Double(u), i);
     }

/*********************************************************************/
    public void insert_i_Weight(int i, int u)
    //inserrer un entier u dans le vecteur des poids à la position  i
    {
       Weight.insertElementAt(new Integer(u), i);
    }

/*********************************************************************/
    public void insert_i_Weight(int i, double u)
    //inserrer un réel u dans le vecteur des poids à la position  i
    {
       Weight.insertElementAt(new Double(u), i);
    }

/*********************************************************************/
    public void insert_i_VC(int i, int u)
```

```
   //inserrer un entier u dans le vecteur VC à la position  i
   {
      VC.insertElementAt(new Integer(u), i);
   }

/*********************************************************************/
   public void insert_i_VP(int i, int u)
   //inserrer un entier u dans le vecteur VP à la position  i
   {
      VP.insertElementAt(new Integer(u), i);
   }

/*********************************************************************/
   public void insert_i_VP_To(int i, int u)
   //inserrer un entier u dans le vecteur VP_To à la position  i
   {
      VP_To.insertElementAt(new Integer(u), i);
   }

/*********************************************************************/
   public void insert_i_VS(int i, int u)
   //inserrer un entier u dans le vecteur VS à la position  i
   {
      VS.insertElementAt(new Integer(u), i);
   }

/*********************************************************************/
   public void insert_i_VS_To(int i, int u)
   //inserrer un entier u dans le vecteur VS_To à la position  i
   {
      VS_To.insertElementAt(new Integer(u), i);
   }

/*********************************************************************/
   public void Affich(int i)
   //  Affichage des différents vecteurs pour l'itération i
   {
        System.out.println("");
        System.out.println("Generation " + i + " :");
        System.out.println("VC " + VC);
        System.out.println("VP " + VP);
        System.out.println("VS " + VS);
        System.out.println("Weight " + Weight);
        System.out.println("Thresh " + Thresh );
        System.out.println("State " + State );
        System.out.println("Actfn " + Actfn );
        System.out.println("VP_To " + VP_To);
        System.out.println("VS_To " + VS_To);
        System.out.println("Link " + Link);
    }

/*********************************************************************/
   public void Affich_Tab()
   //  Affichage
  {
      int g1, g2, g3;
      for(int i=1; i< (Nbr_Neur+1) ; ++i)
      {   g1=Get_VP_To_i(i);
          g2=Get_VP_To_i(i+1);
          if(g1==g2) for(int j=0; j< Nbr_Neur; ++j) {
System.out.print("0"); }
```

```
            if(g1<g2)
            {  g3=Get_VS_To_i(g1);
               for(int j=1; j< (Nbr_Neur+1); ++j)
               {  if(g3==j)
                  {  System.out.print("1");
                        ++g1;
                        if(g1!=g2) g3=Get_VS_To_i(g1);
                     }
                  else System.out.print("0");
               }
            }
            System.out.println(" ");
         }
      }

/*********************************************************************/
   public void Ajout_1_VC_i(int i)
   //  ajout de 1 à l'élément i du vecteur VC
     {
            Integer ObjInt = (Integer)VC.get(i);
      int u = ObjInt.intValue();
      u++;
          VC.set(i,new Integer(u) );
     }

/*********************************************************************/
   public void Ajout_1_VP_i(int i)
   //  ajout de 1 à l'élément i du vecteur VP
     {
      Integer ObjInt = (Integer)VP.get(i);
      int u = ObjInt.intValue();
      u++;
      VP.set(i,new Integer(u) );
     }

/*********************************************************************/
   public void Ajout_1_VP_To_i(int i)
   //  ajout de 1 à l'élément i du vecteur VP_To
     {
      Integer ObjInt = (Integer)VP_To.get(i);
      int u = ObjInt.intValue();
      u++;
      VP_To.set(i,new Integer(u) );
     }

/*********************************************************************/
   public void Ajout_1_VS_i(int i)
   //  ajout de 1 à l'élément i du vecteur VS
     {
      Integer ObjInt = (Integer)VS.get(i);
      int u = ObjInt.intValue();
      u++;
      VS.set(i,new Integer(u) );
     }

/*********************************************************************/
   public void Ajout_1_VS_To_i(int i)
   //  ajout de 1 à l'élément i du vecteur VS_To
     {
      Integer ObjInt = (Integer)VS_To.get(i);
      int u = ObjInt.intValue();
```

```
      u++;
      VS_To.set(i,new Integer(u) );
     }

/*********************************************************************/
   public void Ajout_Src_Dst(int s, int d)
   //  Ajout d'une connexion entre le neurone source s et le neurone
destination d
    {
      Integer ObjInt = (Integer)VP.get(d);
      int d1 = ObjInt.intValue();
      int u=d+1;
      ObjInt = (Integer)VP.get(u);
      int d2 = ObjInt.intValue();
      int w = d2;
      for(int i=d1;i<d2;++i)
          {
          ObjInt = (Integer)VS.get(i);
          int VSi = ObjInt.intValue();
          if( VSi>s) {w=i;}
        }
         insert_i_VS(w, s);
       int g =  Nbr_Neur+1;
       ++d;
       for(int i=d;i<(Nbr_Neur+2);i++)
        { Ajout_1_VP_i(i);
        }
      Ajout_1_VS_i(0);
     }

/*********************************************************************/
   public void Ajout_Src_Dst_To(int d, int s)
   //  Ajout d'une connexion entre le neurone destination d et le
neurone source s
    {
      Integer ObjInt = (Integer)VP_To.get(d);
      int d1 = ObjInt.intValue();
      int u=d+1;
      ObjInt = (Integer)VP_To.get(u);
      int d2 = ObjInt.intValue();
      int w = d2;
      for(int i=d1;i<d2;++i)
          {
          ObjInt = (Integer)VS_To.get(i);
          int VSi = ObjInt.intValue();
          if( VSi>s) {w=i;}
        }
         insert_i_VS_To(w, s);
       int g =  Nbr_Neur+1;
       ++d;
       for(int i=d;i<(Nbr_Neur+2);i++)
        { Ajout_1_VP_To_i(i);
        }
      Ajout_1_VS_To_i(0);
     }

/*********************************************************************/
    public void New_Neur()
    // Ajout d'un neurone au réseau
    {
      ++Nbr_Neur_Courant;
```

```
    }

/*************************************************************************/
    public void New_Layer()
    // Ajout d'une nouvelle couche
    {
      Integer ObjInt = (Integer)VC.get(0);
      int d = ObjInt.intValue();
      ++d;
      VC.set(0,new Integer(d) );
      VC.addElement(new Integer(Nbr_Neur_Courant));
    }

/*************************************************************************/
    public void New_Layer2(int  i)
    // Ajout d'une nouvelle couche avec le neurone courant égale à i
    {
      Integer ObjInt = (Integer)VC.get(0);
      int d = ObjInt.intValue();
      ++d;
      VC.set(0,new Integer(d) );
      VC.addElement(new Integer(i));
    }

/*************************************************************************/
    public void Init_Weight_State_Actfn()
    // Initialisation des poids, des états et des activations
    {
      //Weight
      Integer ObjInt = (Integer)VS.get(0);
      int d = ObjInt.intValue();
      ObjInt = (Integer)VS.get(0);
      int VS0 = ObjInt.intValue();
      Weight.addElement(new Integer(VS0));

      for(int i=1;i<(d+1);i++)
      {
        a1 = ( ( Math.rint(a1*200) ) / 100 ) -1;
        Weight.addElement(new Double(a1));
        a1 = Alea.Gen_Aleat_Double() ;
      }
      //State

      ObjInt = (Integer)VC.get(0);
      int c0 = ObjInt.intValue();
     if(c0>1) {
      ObjInt = (Integer)VC.get(1);
      int c1 = ObjInt.intValue();

      ObjInt = (Integer)VC.get(2);
      int c2 = ObjInt.intValue();

      c2 = c2-c1+1;
      ObjInt = (Integer)VP.get(0);
      int n = ObjInt.intValue();
      State.addElement(new Integer(Nbr_Neur));
      Thresh.addElement(new Integer(Nbr_Neur));
      Actfn.addElement(new Integer(Nbr_Neur));
      for(int i=1;i<(n);i++)
      {
        State.addElement(new Double(1));
```

```
        if(i<c2)
            {
                  Actfn.addElement(new Integer(-1));
                  Thresh.addElement(new Double(-5));
            }
        else
           {
           Actfn.addElement(new Integer(1));
             al = ( Math.rint(al*100) ) / 100;
               Thresh.addElement(new Double(al));
               al = Alea.Gen_Aleat_Double() ;
         };
      }
      ObjInt = (Integer)VS.get(0);
      Nbr_Poids = ObjInt.intValue();
}
    }

/*************************************************************************/
   public void Set_Nbr_Intput()
   // Calcul le nombre de neurones d'entrée
     {
      Integer ObjInt = (Integer)VC.get(1);
      int u1 = ObjInt.intValue();
      ObjInt = (Integer)VC.get(2);
      int u2 = ObjInt.intValue();
      Nbr_Input = u2 -u1;
     }

/*************************************************************************/
   public int Nbr_Input()
   // Renvoit le nombre d'entrées
     {
      return Nbr_Input;
     }

/*************************************************************************/
   public void Set_Nbr_Output()
   // Calcul le nombre de neurones de sorties
     {
      Integer ObjInt = (Integer)VC.get(0);
      int u1 = ObjInt.intValue();
      ObjInt = (Integer)VC.get(u1);
      int u2 = ObjInt.intValue();
      ObjInt = (Integer)VP.get(0);
      int u3 = ObjInt.intValue();
      Nbr_Output = u3-u2;
     }

/*************************************************************************/
   public int Nbr_Output()
   // Renvoit le nombre de sortie
     {
      return Nbr_Output;
     }

/*************************************************************************/
   public void In_State(int r, int c, double d)
      {
      Integer ObjInt = (Integer)VP.get(c);
      int u1 = ObjInt.intValue();
```

```
      ObjInt = (Integer)VC.get(++c);
      int u2 = ObjInt.intValue();
      int u3=0;
      for(int j=u1;j<++u2;++j)
      {
          if(j==r) {u3=j;break;}
      }
      ObjInt = (Integer)VP.get(u3);
      State.set(u3,new Double(d) );
      }

/*********************************************************************/
   public void In_State_Lin(int i, double d)
   // Mise à jour de l'élément i en le remplacant par d dans le
vecteur State
      {
      State.set((i-1),new Double(d) );
      }

/*********************************************************************/
   public int Get_VP_i(int i)
   //Renvoit l'élément i du vecteur VP
     {
      Integer ObjInt = (Integer)VP.get(i);
      return ObjInt.intValue();
     }

/*********************************************************************/
   public int Get_VP_To_i(int i)
   //Renvoit l'élément i du vecteur VP_To
     {
      Integer ObjInt = (Integer)VP_To.get(i);
      return ObjInt.intValue();
     }

/*********************************************************************/
   public int Get_VC_i(int i)
  //Renvoit l'élément i du vecteur VC
     {
      Integer ObjInt = (Integer)VC.get(i);
      return ObjInt.intValue();
     }

/*********************************************************************/
   public int Get_Nbr_Layer()
  //Renvoit le nombre de couches
     {
      Integer ObjInt = (Integer)VC.get(0);
      return ObjInt.intValue();
     }

/*********************************************************************/
   public int Get_VS_i(int i)
   //Renvoit l'élément i du vecteur VS
     {
      Integer ObjInt = (Integer)VS.get(i);
      return ObjInt.intValue();
     }

/*********************************************************************/
   public int Get_VS_To_i(int i)
```

```
  //Renvoit l'élément i du vecteur VS_To
     {
      Integer ObjInt = (Integer)VS_To.get(i);
      return ObjInt.intValue();
     }

/*********************************************************************/
   public double Get_Tresh_i(int i)
   //Renvoit l'élément i du vecteur Tresh
     {
      Double ObjDbl = (Double)Thresh.get(i);
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public double Get_State_i(int i)
  //Renvoit l'élément i du vecteur State
     {
      Double ObjDbl = (Double)State.get(i);
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public int Get_Nbr_Neur()
  //Renvoit le nombre de neurones
      { return Nbr_Neur; }

/*********************************************************************/
   public int Get_Nbr_Weight()
   //Renvoit le nombre de poids
      {
         Integer ObjInt = (Integer)VS.get(0);
         return  ObjInt.intValue();
      }

/*********************************************************************/
   public int Get_Nbr_Poids()
      //Renvoit le nombre de poids
      { return Nbr_Poids; }

/*********************************************************************/
   public double Get_W_i(int i)
   //Renvoit l'élément i du vecteur de poids
     {
      Double ObjDbl = (Double)Weight.get(i);
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public void Set_Nbr_Intput(int i)
   // le nombre d'entrée est égale à i
     {
      Nbr_Input = i;
     }

/*********************************************************************/
   public void Set_Nbr_Output(int i)
   // le nombre de sortie
     {
      Nbr_Output = i;
     }
```

```
/*********************************************************************/
   public void Set_Weight_i(int i, double d)
   // Modifie le poids i avec la valeur d
     {
      Weight.set(i,new Double(d) );
     }

/*********************************************************************/
   public void Set_VC_i(int i, int d)
      // Modifie le i ème élément du vecteur VC par la valeur d
     {
      VC.set(i,new Integer(d) );
     }

/*********************************************************************/
   public void Set_VP_i(int i, int d)
    // Modifie le i ème élément du vecteur VP par la valeur d
     {
      VP.set(i,new Integer(d) );
     }

/*********************************************************************/
   public void Set_VS_i(int i, int d)
    // Modifie le i ème élément du vecteur VS par la valeur d
     {
      VS.set(i,new Integer(d) );
     }

/*********************************************************************/
   public void Set_VS_To_i(int i, int d)
    // Modifie le i ème élément du vecteur VS_To par la valeur d
     {
      VS_To.set(i,new Integer(d) );
     }

/*********************************************************************/
   public void Set_VP_To_i(int i, int d)
   // Modifie le i ème élément du vecteur VP_To par la valeur d
     {
      VP_To.set(i,new Integer(d) );
     }

/*********************************************************************/
   public void Set_Tresh_i(int i, double d)
   // Modifie le i ème élément du vecteur Tresh par la valeur d
     {
      Thresh.set(i,new Double(d) );
     }

/*********************************************************************/
   public void Init_GANN_Miller(int Nbr_Neur, int Nbr_In, int
Nbr_Out)
//   initialisation pour la méthode de Miller
   {
   }

/*********************************************************************/
   public int Get_Link_i(int i)
//   renvoit l'élément i du vecteur Link
     {
```

```
      Integer ObjInt = (Integer)Link.get(i);
      return ObjInt.intValue();
     }

/*********************************************************************/
   public void Do_Link()
//   évalue les éléments du vecteur Link
  {
      Integer ObjInt = (Integer)VS.get(0);
      int u = ObjInt.intValue();

      ObjInt = (Integer)VS.get(0);
      int VS0 = ObjInt.intValue();
      Link.addElement(new Integer(VS0));
      int a;

      for(int i=1; i<(Nbr_Neur+1) ; i++)
      {
         for(int j=1; j< (u+1); j++)
         {
            ObjInt = (Integer)VS.get(j);
            a = ObjInt.intValue();
            if(a==i)
            { Link.addElement(new Integer(j));
            }
         }
      }
  }
}
```

9.1.3.2 Struct_In

9.1.3.2.1 NN_Struct_In

```
package NN.Struct_In;
import java.io.*;
import java.util.*;      // pour StrongTokenizer
import NN.Struct.*;

public class NN_Struct_In
{
//  public static void main (String args[]) throws IOException

  NN_Struct NN;

/*********************************************************************/
  public NN_Struct_In (String nomfich) throws IOException
  //Constructeur pour la lecture d'un réseau de neurone
     {      //String nomfich;
      String ligneRef = "Couche n°:";
      int Num_Layer = 0;
      int Num_Src = 0;
      int Num_Dest = 0;

      //nomfich = "NN/Struct_In/NN_Struct_v1.txt";
      BufferedReader entree = new BufferedReader(new
FileReader(nomfich) );

      // Nombre total de neurones
        String ligneLue = entree.readLine();
```

```
        ligneLue = entree.readLine(); // Lecture nouvelle ligne pour
passer titre
      int Nbr_Neur = Integer.parseInt (ligneLue); // Casting valeur
        ligneLue = entree.readLine();

      NN = new NN_Struct(Nbr_Neur);

      while(true) {
         if(ligneLue == null) break;
         if (ligneLue.equals(ligneRef) ) {
         ligneLue = entree.readLine(); // Lecture valeurs
         Num_Layer = Integer.parseInt (ligneLue); // Casting valeur
         NN.New_Layer();
            }
          else  {
              ligneLue = entree.readLine(); // Lecture nouvelle ligne
pour passer titre
        StringTokenizer tok1 = new StringTokenizer(ligneLue, " " );
            int nv = tok1.countTokens();
            Num_Dest = Integer.parseInt (tok1.nextToken() );
            NN.New_Neur();
            //System.out.println("Neurones Dest " + Num_Dest);
            if (nv > 0)
               {
                 for (int i=2; i<(nv+1); i++)
                  {
                    Num_Src = Integer.parseInt (tok1.nextToken() );
                  //System.out.println("Neurones Src " + Num_Src);
                  NN.Ajout_Src_Dst(Num_Src, Num_Dest);
                  NN.Ajout_Src_Dst_To(Num_Src, Num_Dest);
                  }
                }
            }
           ligneLue = entree.readLine();
      }
      entree.close();
    NN.Init_Weight_State_Actfn();
    NN.Do_Link();
    NN.Set_Nbr_Intput();
    NN.Set_Nbr_Output();
   }

/*********************************************************************/
   public NN_Struct get_Struct()
   // Renvoit le réseau de neurones NN
   {
      return NN;
   }

/*********************************************************************/
   public void Affiche_NN(int u)
   // Affiche le réseau de neurones
   {
      NN.Affich(u);
   }

/*********************************************************************/
   public void Do_Link()
   // Evalue le vecteur Link
   {
      NN.Do_Link();
```

```
    }
}
```

9.1.3.2.2 NN_Weight_Struct_In

```
package NN.Struct_In;
import java.io.*;
import java.util.*;      // pour StrongTokenizer
import NN.Struct.*;

public class NN_Weight_Struct_In
{
  NN_Struct NN;  // Réseau de neurones qui va être construit

/*********************************************************************/
  public NN_Weight_Struct_In (String nomfich) throws IOException
    //Constructeur pour la lecture d'un réseau de neurones au départ
d'un fichier résultats
     {      String nomfich_In;
          int Num = 0;
          double dbl=0;
      int k;

      BufferedReader entree = new BufferedReader(new
FileReader(nomfich) );

        String ligneLue = entree.readLine();   // Lecture nouvelle
ligne pour passer titre Weight
        String ligneLue2 = entree.readLine();
        String ligneLue3 = entree.readLine();   // Lecture nouvelle
ligne pour passer titre Tresh
        String ligneLue4 = entree.readLine();

          StringTokenizer tok1 = new StringTokenizer(ligneLue4, " ");
            int nv = tok1.countTokens();
          NN = new NN_Struct(nv);
          NN.insert_i_Thresh(0, nv);
      NN.insert_i_State(0, nv);
        for (int i=1; i<(nv+1); i++)
          {
              dbl = Double.parseDouble (tok1.nextToken() );
            NN.insert_i_Thresh(i, dbl);
            NN.insert_i_State(i, 1);
      }
          tok1 = new StringTokenizer(ligneLue2, " ");
            nv = tok1.countTokens();
      NN.insert_i_Weight(0, nv);
        for (int i=1; i<(nv+1); i++)
      {
              dbl = Double.parseDouble (tok1.nextToken() );
            NN.insert_i_Weight(i, dbl);
      }

        ligneLue = entree.readLine();   // Lecture nouvelle ligne
pour passer titre
          ligneLue = entree.readLine();
      tok1 = new StringTokenizer(ligneLue, " ");
            nv = tok1.countTokens();
        for (int i=1; i<(nv+1); i++)
      {
              k = Integer.parseInt (tok1.nextToken() );
```

```
            NN.Set_Nbr_Intput(k);
            }

        ligneLue = entree.readLine();    // Lecture nouvelle ligne
pour passer titre
      ligneLue = entree.readLine();
          tok1 = new StringTokenizer(ligneLue, " ");
            nv = tok1.countTokens();
        for (int i=1; i<(nv+1); i++)
      {
            k = Integer.parseInt (tok1.nextToken() );
              NN.Set_Nbr_Output(k);
      }

        ligneLue = entree.readLine();    // Lecture nouvelle ligne
pour passer titre
          ligneLue = entree.readLine();
          tok1 = new StringTokenizer(ligneLue, " ");
            nv = tok1.countTokens();
          NN.Set_VC_i(0, nv);
        for (int i=1; i<(nv+1); i++)
      {
          k = Integer.parseInt (tok1.nextToken() );
              NN.insert_i_VC(i, k);
           }

        ligneLue = entree.readLine();    // Lecture nouvelle ligne
pour passer titre
      ligneLue = entree.readLine();
          tok1 = new StringTokenizer(ligneLue, " ");
            nv = tok1.countTokens();
        for (int i=1; i<(nv+1); i++)
          {
              k = Integer.parseInt (tok1.nextToken() );
          }

        ligneLue = entree.readLine();    // Lecture nouvelle ligne
pour passer titre
      ligneLue = entree.readLine();
          tok1 = new StringTokenizer(ligneLue, " ");
            nv = tok1.countTokens();
          NN.Set_VS_i(0, nv);
        for (int i=1; i<(nv+1); i++)
          {
              k = Integer.parseInt (tok1.nextToken() );
            NN.insert_i_VS(i, k);
          }

        ligneLue = entree.readLine();    // Lecture nouvelle ligne
pour passer titre
      ligneLue = entree.readLine();
          tok1 = new StringTokenizer(ligneLue, " ");
            nv = tok1.countTokens();
        for (int i=1; i<(nv+1); i++)
      {
              k = Integer.parseInt (tok1.nextToken() );
          }

        ligneLue = entree.readLine();    // Lecture nouvelle ligne
pour passer titre
      ligneLue = entree.readLine();
```

```
            tok1 = new StringTokenizer(ligneLue, " ");
              nv = tok1.countTokens();
        NN.Set_VS_To_i(0, nv);
          for (int i=1; i<(nv+1); i++)
            {
                k = Integer.parseInt (tok1.nextToken() );
              NN.insert_i_VS_To(i, k);
            }
              NN.Do_Link();
          entree.close();
        }

/*********************************************************************/
   public NN_Struct get_Struct()
   // Renvoit le réseua de neurone NN
   {
      return NN;
   }

/*********************************************************************/
   public void Affiche_NN(int u)
   // Affiche le réseau de neurones
   {
      NN.Affich(u);
   }
}
```

9.1.3.3 Struct_Out

```
package NN.Struct_Out;
import java.io.*;
import java.util.*;      // pour StrongTokenizer
import NN.Struct.*;

public class NN_Struct_Out
{
  public NN_Struct_Out (NN_Struct NN, String nomfich) throws
IOException
   {
      PrintWriter sortie = new PrintWriter (new FileWriter (nomfich)
);

      int Vc0 = NN.Get_VC_i(0);
      int Vp0 = NN.Get_VP_i(0);
      int Vs0 = NN.Get_VS_i(0);
      int VpTo0 = NN.Get_VP_To_i(0);
      int VsTo0 = NN.Get_VS_To_i(0);

      int t;
      double d;

      sortie.println("Weight");
      for(int i=1;i<(Vs0+1); ++i)
      {
         d=NN.Get_W_i(i);
         sortie.print(d + " ");
      }

      sortie.println("");
      sortie.println("Thresh");
      for(int i=1;i<(Vp0); ++i)
```

```
        {
            d=NN.Get_Tresh_i(i);
            sortie.print(d + " ");
        }

        sortie.println("");
        sortie.println("Nbr_Input");
        t=NN.Nbr_Input();
        sortie.println(t + " ");

        sortie.println("Nbr_Output");
        t=NN.Nbr_Output();
        sortie.println(t + " ");

        sortie.println("VC");
        for(int i=1;i<(Vc0+1); ++i)
        {
            t=NN.Get_VC_i(i);
            sortie.print(t + " ");
        }

        sortie.println("");
        sortie.println("VP");
        for(int i=1;i<(Vp0+1); ++i)
        {
            t=NN.Get_VP_i(i);
            sortie.print(t + " ");
        }

        sortie.println("");
        sortie.println("VS");
        for(int i=1;i<(Vs0+1); ++i)
        {
            t=NN.Get_VS_i(i);
            sortie.print(t + " ");
        }

        sortie.println("");
        sortie.println("VP_To");
        for(int i=1;i<(VpTo0+1); ++i)
        {
            t=NN.Get_VP_To_i(i);
            sortie.print(t + " ");
        }

        sortie.println("");
        sortie.println("VS_To");
        for(int i=1;i<(VsTo0+1); ++i)
        {
            t=NN.Get_VS_To_i(i);
            sortie.print(t + " ");
        }
        sortie.close();
    }

}
```

9.1.3.4 Pattern

```
package NN.Pattern;

import java.io.*;
import java.util.*;      // pour StrongTokenizer

public class NN_Pattern_In
{

  double Data_In[][];
  double Data_Out[][];
  double Data_In_Test[][];
  double Data_Out_Test[][];
  int Nbr_Test;
  int Nbr_Sample;
  int Nbr_Neur_In;
  int Nbr_Neur_Out;

  public NN_Pattern_In (String nomfich) throws IOException

     {
      BufferedReader entree = new BufferedReader(new
FileReader(nomfich) );

 String ligneLue = entree.readLine(); // passer "Nbr de neurones In:"
 ligneLue = entree.readLine(); // Lecture valeurs
 Nbr_Neur_In= Integer.parseInt (ligneLue); // Casting valeur

 ligneLue = entree.readLine(); // passer "Nbr de neurones Out:"
 ligneLue = entree.readLine(); // Lecture valeurs
 Nbr_Neur_Out= Integer.parseInt (ligneLue); // Casting valeur

 ligneLue = entree.readLine(); // passer "Nbr de samples:"
 ligneLue = entree.readLine(); // Lecture valeurs
 Nbr_Sample= Integer.parseInt (ligneLue); // Casting valeur

        ligneLue = entree.readLine(); // passer "Nbr de Test:"
        ligneLue = entree.readLine(); // Lecture valeurs
            Nbr_Test= Integer.parseInt (ligneLue); // Casting valeur

        Data_In = new double [Nbr_Sample][Nbr_Neur_In];
        Data_Out = new double [Nbr_Sample][Nbr_Neur_Out];
        Data_In_Test = new double [Nbr_Test][Nbr_Neur_In];
        Data_Out_Test = new double [Nbr_Test][Nbr_Neur_Out];

      for (int j=0; j< (Nbr_Sample); ++j) {
      ligneLue = entree.readLine(); // Lecture valeurs
           StringTokenizer tok1 = new StringTokenizer(ligneLue, " ");
         int nv = tok1.countTokens();
           for (int i=0; i<(nv); i++) {
            if(i<(Nbr_Neur_In)) {
                  Data_In[j][i] =
Double.parseDouble(tok1.nextToken() );
                 }
            else  {
             Data_Out[j][i-Nbr_Neur_In] =
Double.parseDouble(tok1.nextToken() );
                 }
            }
        }
```

```
      for (int j=0; j< (Nbr_Test); ++j) {
         ligneLue = entree.readLine(); // Lecture valeurs
         StringTokenizer tok1 = new StringTokenizer(ligneLue, " ");
         int nv = tok1.countTokens();
         for (int i=0; i<(nv); i++) {
            if(i<(Nbr_Neur_In)) {
                  Data_In_Test[j][i] =
Double.parseDouble(tok1.nextToken() );
                 }
            else  {
             Data_Out_Test[j][i-Nbr_Neur_In] =
Double.parseDouble(tok1.nextToken() );
                 }
            }
        }
      entree.close();
}

  public double Get_Data_In(int i, int j)
    {
      return  Data_In[i-1][j-1];
    }

  public double Get_Data_Out(int i, int j)
    {
      return  Data_Out[i-1][j-1];
    }

  public double Get_Data_In_Test(int i, int j)
    {
      return  Data_In_Test[i-1][j-1];
    }

  public double Get_Data_Out_Test(int i, int j)
    {
      return  Data_Out_Test[i-1][j-1];
    }

  public int Get_Nbr_Sample()
    {
      return  Nbr_Sample;
    }

  public int Get_Nbr_Test()
    {
      return  Nbr_Test;
    }

  public int Get_Nbr_Neur_In()
    {
      return  Nbr_Neur_In;
    }

  public int Get_Nbr_Neur_Out()
    {
      return  Nbr_Neur_Out;
    }

}
```

9.1.3.5 FeedForward

```
package NN.FeedForward;

import java.lang.Math;
import NN.Struct.*;
import NN.Pattern.*;
import NN.FeedForward.*;
import NN.BackPropagation.*;
import java.io.*;

public class NN_FeedForward
{

boolean Indiv_Actif_Flag = false;
int Indiv_Actif[];   // 1 = entrée active  et 0 = entrée inactive
boolean In_Actif_Flag = false;
int In_Actif[];  // 1 = entrée active  et 0 = entrée inactive

/*********************************************************************/
public NN_FeedForward()
{
}

/*********************************************************************/
public NN_FeedForward(NN_Struct NN, NN_Pattern_In Data)
{
   int Nbr_in = NN.Nbr_Input();
   In_Actif = new int[(Nbr_in+1)];
   for(int i = 1; i< (Nbr_in+1) ; ++i)
   { In_Actif[i] = 1; }

   int Nbr_Indiv = Data.Get_Nbr_Sample();
   Indiv_Actif = new int[(Nbr_Indiv+1)];
   for(int i = 1; i< (Nbr_Indiv+1) ; ++i)
   { Indiv_Actif[i] = 1; }
}

/*********************************************************************/
public double NN_FeedForward_Main(NN_Pattern_In Data, NN_Struct NN,
int Nbr_Iter)
{
double err;
int Nbr_Indiv = Data.Get_Nbr_Sample();
int Nbr_Layer = NN.Get_Nbr_Layer();
  err=0;
  for(int i=0; i<Nbr_Iter; ++i)
  {  err=0;
       for(int u =1; u<(Nbr_Indiv+1); ++u)
       {
            if(Indiv_Actif[u]==1)
            {
                 UpDateInput(NN, Data, u);
                 for(int h = 2; h<(Nbr_Layer);++h)
                 {     UpDateLayer(NN,h);       }
                 err = LessSqrError(NN, Data, u);
                 NN_BackPropagation BP = new NN_BackPropagation(NN);
                 BP.UpDateWeightandThresh(Data, u);
            }
            else
            { --Nbr_Indiv;     }
```

```
        }
        err = 100 * err / Nbr_Indiv;
    }
     return err;
}

/*********************************************************************/
public void UpDate_In_Actif(int w[])
{  In_Actif = w;
   In_Actif_Flag = true;
}

/*********************************************************************/
public void UpDate_Indiv_Actif(int w[])
{  Indiv_Actif = w;
   Indiv_Actif_Flag = true;
}

/*********************************************************************/
public void UpDateInput(NN_Struct old_NN, NN_Pattern_In Data1, int p)
  // Met à jour les entrées du réseau old_NN, sur base de l'élément p
de la base
  // de données des éléments de tests Data1
    {
         NN_Struct new_NN = old_NN;
         int Nbr_in = new_NN.Nbr_Input();
         for(int u=1;u<(Nbr_in+1);++u)
           { double d = Data1.Get_Data_In(p, u);
             new_NN.In_State_Lin(u+1,d);
           }
    }

/*********************************************************************/
   public void UpDateInput_Test(NN_Struct old_NN, NN_Pattern_In
Data1, int p)
   // Calcul de la somme des erreurs (au sens des moindres carrés)
pour le réseau NN
   // pour l'élément p de la base de données des éléments Data1
    {
         NN_Struct new_NN = old_NN;
         int Nbr_in = new_NN.Nbr_Input();
         for(int u=1;u<(Nbr_in+1);++u)
           {
             double d = Data1.Get_Data_In_Test(p, u);
             new_NN.In_State_Lin(u+1,d);
           }
    }

/*********************************************************************/
   public double LessSqrError(NN_Struct NN, NN_Pattern_In Data1, int
p)
    {
        int Nbr_Layer = NN.Get_VC_i(0);
        int d = NN.Get_VC_i(Nbr_Layer) -1;
      int Nbr_Out = NN.Nbr_Output();
      double r=0;
      for(int i=1;i<(Nbr_Out+1);++i)
      {
           int u = i+d;
           double d2 = NN.Get_State_i(u);
           double d1 = Data1.Get_Data_Out(p, i);
```

```
            r = ( (d1-d2)*(d1-d2) ) +r;
        }
        r = r / Nbr_Out;
        return r;
    }

/*********************************************************************/
   public double LessSqrError_Test(NN_Struct NN, NN_Pattern_In Data1,
int p)
// Calcul de la somme des erreurs (au sens des moindres carrés) pour
le réseau NN
// pour l'élément p de la base de données de tests des éléments Data1

    {
        int Nbr_Layer = NN.Get_VC_i(0);
        int d = NN.Get_VC_i(Nbr_Layer) -1;
      int Nbr_Out = NN.Nbr_Output();
      double r=0;
      for(int i=1;i<(Nbr_Out+1);++i)
      {
           int u = i+d;
           double d2 = NN.Get_State_i(u);
           double d1 = Data1.Get_Data_Out_Test(p, i);
        r = ( (d1-d2)*(d1-d2) ) +r;
      }
      r = r / Nbr_Out;
      return r;
    }

/*********************************************************************/
   public void UpDateLayer(NN_Struct old_NN, int p)
 //  Calcul de la couche p du réseau sur base des différentes entrées
    {
         NN_Struct new_NN = old_NN;
         int Nbr_Neur = new_NN.Get_Nbr_Neur();
         int u0 = new_NN.Get_VC_i(0);
         int u1 = new_NN.Get_VC_i(p);
         int u2;
         if(u0>p)
              {          // cas pas dernière couche
                u2 = new_NN.Get_VC_i(p+1);
              }
         else
            { u2 = new_NN.Get_VP_i(0);  }
         for(int u=u1;u<u2;++u)
       {
           double act = new_NN.Get_Tresh_i(u);
           int u3 = u+1;
               int u5 = new_NN.Get_VP_i(u);
               int u6 = new_NN.Get_VP_i(u3);
               for(int k=u5;k<u6;++k)
                  {     double w = new_NN.Get_W_i(k);
                        int n1 = new_NN.Get_VS_i(k);
                        double st = new_NN.Get_State_i(n1);
                        act += st*w;
                  }
               double d1 =  (1.0 / ( 1.0 + Math.exp(0.0 - act) ) );
               new_NN.In_State_Lin(u3, d1);
       }
    }
}
```

9.1.3.6 BackPropagation

```
package NN.BackPropagation;

import java.lang.Math;
import java.util.Vector;
import NN.Struct.*;
import NN.Pattern.*;
import java.io.*;

public class NN_BackPropagation
{
   double EPSILON = 0.5;
   double MOMENTUM = 0.25;
   Vector Deriv1 = new Vector();     //* for holding dE/dy
   Vector Deriv2 = new Vector();     //* for holding dE/ds
   Vector ThreshError = new Vector();     //* for Error on Thresh
   Vector WeightError = new Vector();     //* for Error on Weight
   Vector Prev_ThreshError = new Vector();     //* for Error on Thresh
   Vector Prev_WeightError = new Vector();     //* for Error on Weight

   int Nbr_Neur;
   NN_Struct NN;

/*********************************************************************/
   public void Set_Epsilon(double Eps)
   // Mise à jour de la valeur de EPSILON
   {   EPSILON = Eps;
   }

/*********************************************************************/
   public void Set_Momentum(double Mmt)
   // Mise à jour de la valeur du MOMENTUM
   {   MOMENTUM = Mmt;
   }

/*********************************************************************/
   public NN_BackPropagation(NN_Struct NN_Old)
   // effectue une BP sur le réseau NN_Old
   {
      NN = NN_Old;
      Nbr_Neur = NN.Get_Nbr_Neur();
      Deriv1.addElement(new Integer(Nbr_Neur));
      Deriv2.addElement(new Integer(Nbr_Neur));
    ThreshError.addElement(new Integer(Nbr_Neur));
    Prev_ThreshError.addElement(new Integer(Nbr_Neur));

      for(int i=1; i < (Nbr_Neur + 1); ++i)
        {
          Deriv1.addElement(new Double(0));
          Deriv2.addElement(new Double(0));
          ThreshError.addElement(new Double(0));
          Prev_ThreshError.addElement(new Double(0));
      }

      int np = NN.Get_Nbr_Poids();
    WeightError.addElement(new Integer(np));
    Prev_WeightError.addElement(new Integer(np));
      for(int i=1; i < (np + 1); ++i)
      {
          WeightError.addElement(new Double(0));
```

```
            Prev_WeightError.addElement(new Double(0));
        }
 }

/*********************************************************************/
   public double Get_ThreshError_i(int i)
   // renvoit l'erreur sur l'offset de l'itération courante
     {
      Double ObjDbl = (Double)ThreshError.get(i);
          return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public double Get_Prev_ThreshError_i(int i)
   // renvoit l'erreur sur l'offset de l'itération précédente
     {
            Double ObjDbl = (Double)Prev_ThreshError.get(i);
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public double Get_WeightError_i(int i)
   // renvoit l'erreur sur les poids des connexions de l'itération
courante
     {
      Double ObjDbl = (Double)WeightError.get(i);
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public double Get_Prev_WeightError_i(int i)
    // renvoit l'erreur sur les poids des connexions de l'itération
précédente
     {
      Double ObjDbl = (Double)Prev_WeightError.get(i);
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public double Get_Deriv1_i(int i)
   // renvoit l'élément i du vecteur Deriv 1
     {
      Double ObjDbl = (Double)Deriv1.get(i);
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
   public double Get_Deriv2_i(int i)
   // renvoit l'élément i du vecteur Deriv 2
     {  Double ObjDbl = (Double)Deriv2.get(i);
      ObjDbl.doubleValue() ) ;
      return ObjDbl.doubleValue();
     }

/*********************************************************************/
  public void GetDerivs(NN_Pattern_In Data_Out, int p)
  // calcul de Deriv1 et de Deriv2
  {
    int Nbr_Out = NN.Nbr_Output();
    int Nbr_Layer = NN.Get_VC_i(0);
    int d3 = NN.Get_VC_i(Nbr_Layer) -1;
```

```
    int pp1 = NN.Get_VC_i(Nbr_Layer-1);
    int pp2 = NN.Get_VC_i(Nbr_Layer);

    for(int i=1;i<(Nbr_Out+1);++i)
    {
      double d1 = Data_Out.Get_Data_Out(p, i);
      int u = i+d3;
        double d2 = NN.Get_State_i(u);
      d2 = d1-d2;
      Deriv1.set(u, new Double(d2) );
     }

     for(int i=1;i<(Nbr_Out+1);++i)
     {
       int u = i+d3;

       Double ObjDbl = (Double)Deriv1.get(u);
       double e = ObjDbl.doubleValue();

       double s = NN.Get_State_i(u);
         s = e*s*(1-s);
       Deriv2.set(u, new Double(s) );
     }

     for(int i= (Nbr_Layer-1); i>1; --i)
     {
      int p2 = NN.Get_VC_i(i+1);
      int p1= NN.Get_VC_i(i);

  if(i==(Nbr_Layer-1) ) { p1 = p2 - NN.Nbr_Output(); } // cas ou la
dernière couche a plus de neurone qu'il n'y a de neurones de sortie
       else { p1 = NN.Get_VC_i(i); }
        for(int Node=p1; Node < p2; ++Node)
          {
            int u1 = NN.Get_VP_To_i(Node);
              int u2 = NN.Get_VP_To_i(Node+1);
            double der1 =0;
            for(int ToNode=u1; ToNode < u2; ++ToNode)
              {
                int u7 = NN.Get_VS_To_i(ToNode);
              double d1 = NN.Get_State_i(u7);
              int u8 = NN.Get_Link_i(ToNode);
              double wi = NN.Get_W_i(u8);
              double der2 = Get_Deriv2_i(u7);
                der1= der2*wi+der1;
               }
             Deriv1.set(Node, new Double(der1) );
           }
         for(int Node=p1; Node < p2; ++Node)
           {
             double der1 = Get_Deriv1_i(Node);
               double State1 =   NN.Get_State_i(Node);
             double der2 = der1 * State1 * (1- State1);
             Deriv2.set(Node, new Double(der2) ) ;
         }
}
}

/*****************************************************************/
public NN_Struct UpDateWeightandThresh(NN_Pattern_In Data_Out, int p)
 // Update les valeurs des poids et des Offsets
```

```
{   GetDerivs(Data_Out, p);
    int Nbr_Layer = NN.Get_VC_i(0);
      for(int m=2; m<(Nbr_Layer+1); ++m)
      {
         int u1 = NN.Get_VC_i(m);
         int u2;
         if(m<Nbr_Layer)
       {          // cas pas dernière couche
          u2 = NN.Get_VC_i(m+1);
       }
         else
       {
        u2 = NN.Get_VP_i(0);
       }
         for(int u=u1;u<u2;++u)
       {
            int u3 = u+1;
              double der2 = Get_Deriv2_i(u);
              ThreshError.set(u, new Double(der2));

              int u5 = NN.Get_VP_i(u); //
              int u6 = NN.Get_VP_i(u+1);  //
              for(int k=u5;k<u6;++k)
                 {
                    int w = NN.Get_VS_i(k);
                    double s = NN.Get_State_i(w);
                    s = s * der2;
              WeightError.set(k, new Double(s));
                 }
       }
      }

    int y = NN.Get_VC_i(2);
      Nbr_Neur = NN.Get_Nbr_Neur();

    for(int u=y;u<(Nbr_Neur+1);++u)
      {
      double e =  NN.Get_Tresh_i(u);
      double h = (EPSILON * Get_ThreshError_i(u));
      double d1 = Get_Prev_ThreshError_i(u);
      double de = (EPSILON * Get_ThreshError_i(u)) + (d1 * MOMENTUM);
            Prev_ThreshError.set(u, new Double(de));
            e = de + e;
            NN.Set_Tresh_i(u, e);
      }

      int np = NN.Get_Nbr_Poids();
    for(int u=1;u<(np+1);++u)
      {
     double e =  NN.Get_W_i(u);
     double d1 = Get_Prev_WeightError_i(u);
     double de = (EPSILON * Get_WeightError_i(u)) +  (d1 * MOMENTUM);
            e =  de + e;
        Prev_WeightError.set(u, new Double(de));

            NN.Set_Weight_i(u, e);
      }

      return NN;
      }
}
```

9.1.4 GA_NN

9.1.4.1 Opt_Struct

9.1.4.1.1 NN_Initialisation

```
package Ga_Nn.Opt_Struct;
import Tools.Gen_Aleat.Gen_Aleat;

public class NN_Initialisation
{

/*********************************************************************/
public NN_Initialisation(int p[], int Nbr_Neur, int Nbr_In, int
Nbr_Out)
   {
      Gen_Aleat g = new Gen_Aleat();

    int alpha;
    int r, u, t, tt;
      int rr=0;
      t = Nbr_In+1;
      tt = Nbr_Neur - Nbr_Out;
      for (int i = 1; i< (Nbr_Neur+1); ++i)
      {  if(i<t) u=t;
         else u=i+1;
         for (int j = 1; j< (Nbr_Neur+1); ++j)
         { if ( (j<u) || (i>tt) ) {   p[rr] =0; }
             else {   p[rr] = (int)g.Gen_Aleat_Double(0, 2 );  }
             ++rr;
         }
      }
   }
}
```

9.1.4.1.2 NN_Struct_Modif

```
package Ga_Nn.Opt_Struct;
import NN.Struct.*;

public class NN_Struct_Modif
{

   private int tab[][];
   private int tab_New[][];
   private NN_Struct NN;
   private NN_Struct NN_New;
   int Nbr_From[];
   int Num_Couche[];
   int Temp_Couche[];

/*********************************************************************/
   public NN_Struct_Modif(int t[][], int Nbr_Neur)
   // Création des connexions du réseau au départ d'une matrice de
connexions stockée dans un tableau t
   {
      tab = new int[Nbr_Neur][Nbr_Neur];
      int Num_Src;
      int Num_Dest;
```

```
      NN = new NN_Struct(Nbr_Neur);

      Nbr_From = new int[Nbr_Neur];
      Num_Couche = new int[Nbr_Neur];
      Temp_Couche = new int[Nbr_Neur];
      for(int i=0; i< Nbr_Neur; ++i)
      {  Temp_Couche[i]=i;
         for(int j=0; j< Nbr_Neur; ++j)
          {       NN.New_Neur();
            tab[i][j]=t[i][j];
            if(t[i][j]==1)
            { ++Nbr_From[j];
              Num_Src=i+1;
              Num_Dest=j+1;
                NN.Ajout_Src_Dst(Num_Src, Num_Dest);
              NN.Ajout_Src_Dst_To(Num_Src, Num_Dest);
            }
          }
      }
        NN.Do_Link();
      NN_Struct_Modif_Main(Nbr_Neur);
   }

/*********************************************************************/
   public NN_Struct_Modif(int t[], int Nbr_Neur)
 // Création des connexions du réseau au départ d'un vecteur de
connexions t
   {
      tab = new int[Nbr_Neur][Nbr_Neur];
      int Num_Src;
      int Num_Dest;
      NN = new NN_Struct(Nbr_Neur);

      Nbr_From = new int[Nbr_Neur];
      Num_Couche = new int[Nbr_Neur];
      Temp_Couche = new int[Nbr_Neur];
      int m=0;
      for(int i=0; i< Nbr_Neur; ++i)
      {  Temp_Couche[i]=i;
         for(int j=0; j< Nbr_Neur; ++j)
          {
            tab[i][j]=t[m];
            NN.New_Neur();
            if(t[m]==1)
            {
               ++Nbr_From[j];
              Num_Src=i+1;
              Num_Dest=j+1;
                NN.Ajout_Src_Dst(Num_Src, Num_Dest);
              NN.Ajout_Src_Dst_To(Num_Src, Num_Dest);
            }
            ++m;
          }
         //System.out.println("");
      }
      NN_Struct_Modif_Main(Nbr_Neur);
   }

/*********************************************************************/
   private void NN_Struct_Modif_Main(int Nbr_Neur){
```

```
//VP, VS, VP_To et VS_To ok pour le réseau NN non classé par couche

      //NN_New sera le nouveau réseau classé par couche

      //recherche des numéros de couches des différentes colonnes
(cad des différents neurones)

// Réordonnancement  des colonnes pour minimiser le nombre de couches
      int g, u1, u2, u3, temp1, f, x, s;
      int Num_Couche_Val =1;
      for(int i=1; i<(Nbr_Neur+1); ++i)
      {   u1= NN.Get_VP_i(i);
            u2= NN.Get_VP_i(i+1);
          u3 = u2-u1;
          temp1 =0;
          f=0;
          if(u3==0)
          {  if(Num_Couche_Val==1) Num_Couche[i-1]=1;
             else Num_Couche[i-1]=Num_Couche[i-2];
          }
          else
          {  for(int j=u1;j<u2; ++j)
             {  x= NN.Get_VS_i(j);
              s = Num_Couche[x-1];
              if(s>0) { if( (s+1)>f ) f=s+1;
                        ++temp1; }
              }
              if(u3==temp1)  Num_Couche[i-1]=f;
           };
      }

     //Temp_Couche variable temporaire de classement de colonnes
      int gg;
      for(int i=1; i<Nbr_Neur; ++i)
      {  if(Num_Couche[i] < Num_Couche[i-1])
         {        for(int j=(i-1); j>0; --j)
            {   if(Num_Couche[j] <= Num_Couche[i] )
                {  gg=Temp_Couche[i];
                 Temp_Couche[i]= Temp_Couche[j+1];
                 Temp_Couche[j+1]=gg;
                 j=0;
                }
            }
         }
      }

      //Création d'un nouveau réseau NN_New
      NN_New = new NN_Struct(Nbr_Neur);

    //création de VP, VS, VP_To et VS_To pour le nouveau réseau classé
      int e, Num_Src, Num_Dest;
      for(int j=0; j< Nbr_Neur; ++j)
      {  e = Temp_Couche[j];
         for(int i=0; i< Nbr_Neur; ++i)
          {       NN_New.New_Neur();
            if(tab[i][e]==1)
            { Num_Src=i+1;
              Num_Dest=j+1;
           NN_New.Ajout_Src_Dst( Num_Src, Num_Dest);
              NN_New.Ajout_Src_Dst_To(Num_Src, Num_Dest);
            }
```

```
            }
        }
          NN_New.Do_Link();

          // Update de Vc
        int z1, dz;
        int z2=0;
          NN_New.Ajout_1_VC_i(0);
          NN_New.Set_VC_i(0, 0);
        for(int i=0; i< Nbr_Neur; ++i)
        {  z1 = Temp_Couche[i];
           z1 = Num_Couche[z1];
           dz = z1-z2;
           if(dz>0) NN_New.New_Layer2(i+1);
           z2=z1;
        }
        NN_New.Init_Weight_State_Actfn();
    }

/*************************************************************************/
    public NN_Struct Get_NN_New()
    // Renvoit le nouveau réseau
      {
           return NN_New;
      }
}
```

9.1.4.1.3 NN_Struct_Modif

```
package Ga_Nn.Opt_Struct;

public class NN_CrossOver
{
   public NN_CrossOver(int p1[], int p2[], int v, int Nbr_Neur)
   {  // i = 1 jusque nbr de neurones
      int temp_p1, temp_p2, ii;
      for (int i = 0; i<Nbr_Neur; i++)
      {  ii = i * Nbr_Neur + v-1 ;
         if(p1[ii]==1) { temp_p1 = 1;  } else { temp_p1 = 0;  };
         if(p2[ii]==1) { temp_p2 = 1;  } else { temp_p2 = 0;  };
         p1[ii] = temp_p2;
         p2[ii] = temp_p1;
      }

   }
}
```

9.1.4.1.4 NN_Mutation

```
package Ga_Nn.Opt_Struct;
import Tools.Gen_Aleat.Gen_Aleat;

public class NN_Mutation
{
   public NN_Mutation(int p[], int Nbr_Neur, int Nbr_In, int Nbr_Out)
   {
      Gen_Aleat g = new Gen_Aleat();

      int m1, m2, m3;
      m1 = (Nbr_Neur-Nbr_In)*Nbr_In;
```

```
      m2=0;
      for(int i = Nbr_Out; i< (Nbr_Neur-Nbr_In); ++i)
      {  m2=m2+i; }
      m3 = m1+m2;

      int alpha = (int)g.Gen_Aleat_Double(1, (m3+1) );

      int r, u, t;
      int rr=1;
      t = Nbr_In+1;
      for (int i = 1; i< (Nbr_Neur+1); ++i)
      {  if(i<t) u=t;
         else u=i+1;
         for (int j = u; j< (Nbr_Neur+1); ++j)
           {  r=(j+(((i-1)*Nbr_Neur)));
                if( (rr== alpha) )
                { if (p[r-1]==1) p[r-1]=0;
                  else p[r-1]=1;
                }
              ++rr;
           }
       }
  }
}
```

9.1.4.2 Opt_Weight

9.1.4.2.1 NN_Weight_Modif

```
package Ga_Nn.Opt_Weight;
import NN.Struct.*;

public class NN_Weight_Modif
{
/*********************************************************************/
public NN_Weight_Modif(NN_Struct NN, double t[])
//Permet d'encoder dans le réseau NN, les poids et le offset stockés
dans le vecteur de réels t[]
{
      int u1, u2, u3;
      int Nbr_Neur = NN.Get_Nbr_Neur();
      int Nbr_Weight = NN.Get_VS_i(0);
      int m=0;
      int w=1;
      for(int i=1; i<(Nbr_Neur+1); ++i)
      {   u1= NN.Get_VP_i(i);
        u2= NN.Get_VP_i(i+1);
          for(int j=u1; j<u2; ++j)
          {   u3 = NN.Get_VS_i(j);
                NN.Set_Weight_i(w, t[m]);
                ++w;
                ++m;
          }
          NN.Set_Tresh_i(i, t[m]);
          ++m;
    }
}

/*********************************************************************/
public void NN_Weight_Modif_Back(NN_Struct NN, double t[])
```

```
//Permet d'encoder dans le vecteur de réels t[] les poids et les
offsets stockés dans le réseau NN
{
      int u1, u2, u3;
      int Nbr_Neur = NN.Get_Nbr_Neur();
      int Nbr_Weight = NN.Get_VS_i(0);

      int m=0;
      int w=1;
      for(int i=1; i<(Nbr_Neur+1); ++i)
      {   u1= NN.Get_VP_i(i);
        u2= NN.Get_VP_i(i+1);
          for(int j=u1; j<u2; ++j)
          {   t[m] = NN.Get_W_i(w);
                  ++w;
                  ++m;
          }
          t[m] = NN.Get_Tresh_i(i);
          ++m;
    }
}
}
```

9.1.4.2.2 NN_CrossOver

```
package Ga_Nn.Opt_Weight;
import NN.Struct.*;

public class NN_CrossOver
{

  NN_Struct NN;

/*********************************************************************/
   public NN_CrossOver(NN_Struct NN1)
   // affecte le réseau NN1 au réseau NN
   {
      NN = NN1;
   }

/*********************************************************************/
   public void NN_CrossOver_Node(int N, double t1[], double t2[])
      // Effectue un CrossOver des N ième noeuds des 2 parents stoqué
dans les 2 vecteurs de réels t1[] et t2[]
   {    //avec N entre 1 et 6 pour XOr_v01
      int u1, u2,u3;
      double d;
      u1= NN.Get_VP_i(N);
      u3= NN.Get_VP_i(N+1);
      u3=u3-u1;
    u2= u1-1+N-1;
    for(int j=(u2); j<(u2+u3+1); ++j)
        {   d = t1[j];
            t2[j]=t1[j];
            t1[j]=d;
        }
    }

}
```

9.1.5 Tools

9.1.5.1 Tri

```
package Tools.Tri;
import java.util.*;

public class Tri{

    public LinkedList l;
    public LinkedList l_pos;
    public LinkedList l_pos1;
    public int[] tab2;
        public int[] tab3;

    public Tri(int[] tab){

        /* tri par insertion dans une liste liée */
        l = new LinkedList();
        l_pos = new LinkedList();
        l_pos1 = new LinkedList();

        boolean ok=false;
        l.add(new Integer(tab[0]));
        l_pos.add(new Integer(1));

        tab2 = new int[tab.length];
        tab3 = new int[tab.length];


        for (int i=1; i<tab.length; i++){
            for (int j=0; j<l.size(); j++){
                if (tab[i] <= ((Integer)l.get(j)).intValue()){
                    l.add(j, new Integer(tab[i]));
                    l_pos.add(j, new Integer(i+1));
                    ok=true;
                    tab2[j]=i;
                    break;
                }
            }
            if(!ok){
                l.addLast(new Integer(tab[i]));
                l_pos.addLast(new Integer(i+1));
            }
            ok=false;
        }
    }

    public Tri(double[] tab){

        /* tri par insertion dans une liste liée */
        l = new LinkedList();
        l_pos = new LinkedList();
        tab2 = new int[tab.length];
        tab3 = new int[tab.length];

        boolean ok=false;
        l.add(new Double(tab[0]));
        l_pos.add(new Integer(1));
        for (int i=1; i<tab.length; i++){
            for (int j=0; j<l.size(); j++){
```

```
                if (tab[i] <= ((Double)l.get(j)).doubleValue()){
                    l.add(j, new Double(tab[i]));
                    l_pos.add(j, new Integer(i+1));
                    tab2[i] = i-j;
                    ok=true;
                    break;
                }
            }
            if(!ok){
                l.addLast(new Double(tab[i]));
                l_pos.addLast(new Integer(i+1));
            }
            ok=false;
        }

        for (int i=0; i<(tab3.length); i++){
          Integer Obj = (Integer)l_pos.get(i);
          int d1  = (int)Obj.intValue();
            tab3[d1-1]=i;
        }

   }


    public LinkedList Get_l()
      { return l;
      };

    public LinkedList Get_l_pos()
      { return l_pos;
      };

      public int[] Get_resu()
      { return tab3;
      };

}
```

9.1.5.2 Gen_Aleat

```
package Tools.Gen_Aleat;
import java.util.Random;
import java.lang.Math;

public class Gen_Aleat {

static protected Random genAleatoire = new Random();

/************************************************/
public double Gen_Aleat_Double()
  {
    return genAleatoire.nextDouble();
  }

/************************************************/
public double Gen_Aleat_Double(double i, double j)
  {
    double l;
    l = genAleatoire.nextDouble();
    l = i + (j-i)*l;
    return l;
```

```
  }

/************************************************/
public double Gen_Aleat_Double_Exp_Dist()
  {
    double d;
    double sign;
    d = genAleatoire.nextDouble();
//    System.out.print("  " +d) ;
    d = Math.log(d);
//    System.out.print("  " +d) ;

    sign = genAleatoire.nextDouble();
    if (sign<0.5) d = (-1)*(d);
//    System.out.println("  " +d) ;
    return d;
  }

/************************************************/
public double Gen_Aleat_Double_Exp_Dist2()
  {
    double d;
    double sign;
    d = genAleatoire.nextDouble();
//    System.out.print("  " +d) ;
    d = (-Math.log(d)-20)/20+1;
//    System.out.print("  " +d) ;

    sign = genAleatoire.nextDouble();
    if (sign<0.5) d = (-1)*(d);
//    System.out.println("  " +d) ;
    d=d+0.5;
    return d;
  }

}
```

9.1.6 Test

9.1.6.1 NN_BackPropagation

```
package Test.NN_BackPropagation;
import NN.Struct_In.*;
import NN.Struct_Out.*;
import NN.Struct.*;
import NN.FeedForward.*;
import NN.BackPropagation.*;
import NN.Pattern.*;
import Tools.Gen_Aleat.Gen_Aleat;
import java.util.Date;
import Ga_Nn.Opt_Weight.*;
import java.io.*;

public class NN_BackPropagation_Test
{
   static long now_old  = System.currentTimeMillis();

    public static void main (String args[])
{
      Gen_Aleat g = new Gen_Aleat(); // élément pour génération
aléatoire
```

```
      double Epsilon = 0.75;
      double Momentum = 0.25;

      //String title = "Xor_v01";
      //String title = "Xor_v02";
      //String title = "Cancer_v01";
      //String title = "Cancer_v02";
      //String title = "Cancer_v03";
      //String title = "X1_Mult_X2_v01";
      //String title = "Symetrie_v01";
        //String title = "Symetrie_v02";
      //String title = "Encod_Decod_v01";
      String title = "Iris_v01";

      String Nomfich_Out = "Samples/"  + title  + "/Resu_v1.txt";

      NN_Struct_Out  NN_Out;

  try
  {
        String Nomfich = "Samples/"  + title  + "/Struct_in.txt";
      NN_Struct_In  NN_In = new NN_Struct_In(Nomfich);
      NN_Struct NN = NN_In.get_Struct();

      int Nbr_Chrom = NN.Get_Nbr_Weight() + NN.Get_Nbr_Neur();
      double [] Chrom = new double[Nbr_Chrom];

      for(int i=0; i<Nbr_Chrom ; i++)
          {  Chrom[i] = g.Gen_Aleat_Double(-1 , 1 ) ;
          }

         NN_Weight_Modif NN_W = new NN_Weight_Modif(NN, Chrom);

//          String Nomfich = "Samples/"  + title  +
"/Weight_Struct_In.txt";
//          NN_Weight_Struct_In  NN_In = new
NN_Weight_Struct_In(Nomfich);
//          NN_Struct  NN = NN_In.get_Struct();

         String File_Data = "Samples/"  + title  + "/Pattern.txt";

         NN_Pattern_In Data_Out = new NN_Pattern_In(File_Data);
         NN_FeedForward Data0 = new NN_FeedForward();
           NN_BackPropagation Data1 = new NN_BackPropagation(NN);
         Data1.Set_Epsilon(Epsilon);
         Data1.Set_Momentum(Momentum);

         double s = 0;
         double err=0;
         int Nbr_Indiv = Data_Out.Get_Nbr_Sample();
         int Nbr_Layer = NN.Get_Nbr_Layer();

       for(int i=0; i<Nbr_Chrom ; i++)
       {  Chrom[i] = g.Gen_Aleat_Double(-1 , 1 ) ;     }
       NN_W = new NN_Weight_Modif(NN, Chrom);
       s=0;
       for(int i=1; i<100000; ++i)
       {    err=0;
            for(int u=1;u< (Nbr_Indiv+1); ++u)
            {  Data0.UpDateInput(NN, Data_Out, u);
```

```
                for (int h=2; h<(Nbr_Layer+1);++h)
              {  Data0.UpDateLayer(NN,h);        }
              err = Data0.LessSqrError(NN, Data_Out, u)+err;
              Data1.UpDateWeightandThresh(Data_Out, u);
           }
           s++;
           err = 100* err / Nbr_Indiv;
           long now = System.currentTimeMillis();
           now = (now - now_old)/1000;
                 if (s == 100)  { System.out.println("  " + i + "  "
+ now + " " + err);  s = 0;      NN_Out = new
NN_Struct_Out(NN,Nomfich_Out);          }
        }
//     System.out.println("Chrom[0] : " + Chrom[0] + "Chrom[1] : " +
Chrom[1] + " err : " + err);
     NN_Out = new NN_Struct_Out(NN,Nomfich_Out);
   }
//   NN.Affich(1);


  catch (IOException e)
  {
  }


   }
}
```

9.1.6.2 Ga_NN_Opt_Learn

```
package Test.Ga_NN_Opt_Learn;
import Ga.Population.*;

public class Main_GA_NN_BP_Param {
   public static void main(String[] args) {

  // Variables pour l'individu
   int iter_j = 20;
   String title = "Xor_v02";
   int iter_BP = 100;
      //String title = "Xor_v01";
      //String title = "Cancer_v01";
      //String title = "Cancer_v02";
      //String title = "Cancer_v03";
      //String title = "X1_Mult_X2_v01";
      //String title = "Symetrie_v01";

   // Variables pour la population
   int Nbr_Indiv = 100;  // inf 10
   double Taux_Crois = 0.60; // inf 0, sup 1
   double Taux_Mut = 0.30; // inf 0, sup 1
   boolean Elitis = true;  // true ou false
   double Taux_Elitis = 0.85; // inf 0, sup 1
   boolean Simu_Anneal = true; //  true ou false
   boolean Scal = true; //  true ou false
   String Scal_Type = "Position";  // "Position", "Lin" ou "Exp"
   int Nbr_Gener_GA = 100; // inf 10
   int Aff_Level = 1; // inf 1, sup 5

   // Main
     Popu1_Ga_NN_BP_Param  P = new  Popu1_Ga_NN_BP_Param();
```

```
      P.Set_Param(iter_j , iter_BP, title);

      P.Affichage_Level(Aff_Level);
      P.ModifNombrePop(Nbr_Indiv);
      P.ModifTauxCrois(Taux_Crois);
      P.ModifTauxMut(Taux_Mut);

      if(Scal == true)       P.Scaling(Scal_Type);
      if(Simu_Anneal == true)       P.Simulated_Annealing();
      if(Elitis == true) P.Elitisme_Set(Taux_Elitis);

      P.Initialisation();

      for(int i=0;i<100;i++) {
         P.Evaluation();
         P.Affiche();
         P.Selection();
         P.Croisement();
         P.Mutation();
       }

         P.Evaluation();
         P.Affiche();

   }
}
```

9.1.6.3 Ga_NN_Opt_Weight

```
package Test.Ga_NN_Opt_Weight;
import Ga.Population.*;

public class Main_GA_NN_Davis {
   public static void main(String[] args) {

   // Variables pour l'individu
   boolean flag_BP = true;
   String title = "Iris_v01";
//   String title = "Xor_v02";
   int iter_BP = 100;
      //String title = "Xor_v01";
      //String title = "Cancer_v01";
      //String title = "Cancer_v02";
      //String title = "Cancer_v03";
      //String title = "X1_Mult_X2_v01";
      //String title = "Symetrie_v01";

   // Variables pour la population
   int Nbr_Indiv = 200;  // inf 10
   double Taux_Crois = 0.60; // inf 0, sup 1
   double Taux_Mut = 0.30; // inf 0, sup 1
   boolean Elitis = true;  // true ou false
   double Taux_Elitis = 0.79; // inf 0, sup 1
   boolean Simu_Anneal = true; //  true ou false
   boolean Scal = true; //  true ou false
   String Scal_Type = "Lin";  // "Position", "Lin" ou "Exp"
   int Nbr_Gener_GA = 50000; // inf 10
   int Aff_Level = 1; // inf 1, sup 5

   // Main
     Popu1_Ga_NN_Davis P = new  Popu1_Ga_NN_Davis();
```

```
      P.Set_Param(flag_BP, iter_BP, title);

      P.Affichage_Level(Aff_Level);
      P.ModifNombrePop(Nbr_Indiv);
      P.ModifTauxCrois(Taux_Crois);
      P.ModifTauxMut(Taux_Mut);

      if(Scal == true)       P.Scaling(Scal_Type);
      if(Simu_Anneal == true)       P.Simulated_Annealing();
      if(Elitis == true) P.Elitisme_Set(Taux_Elitis);

      P.Initialisation();

      for(int i=0;i<Nbr_Gener_GA ;i++) {
         P.Evaluation();
         P.Affiche();
         P.Selection();
         P.Croisement();
         P.Mutation();
       }
   }
}
```

9.1.6.4 Ga_NN_Opt_Struct

```
package Test.Ga_NN_Opt_Struct;
import Ga.Population.*;

public class Main_GA_NN_Miller {
   public static void main(String[] args) {

   // Variables pour l'individu
   int Nbr_Neur =7;
   int Nbr_In = 2;
   int Nbr_Out = 1;
   int iter_j = 20;
   int iter_BP = 500;
   String fichier = "Xor_v02";
             String title = "Xor_v02";
            //String title = "Cancer_v01";
            //String title = "Cancer_v02";
            //String title = "Cancer_v03";
            //String title = "X1_Mult_X2_v01";
            //String title = "Symetrie_v01";

   // Variables pour la population
   int Nbr_Indiv = 10;  // inf 10
   double Taux_Crois = 0.60; // inf 0, sup 1
   double Taux_Mut = 0.30; // inf 0, sup 1
   boolean Elitis = true;  // true ou false
   double Taux_Elitis = 0.89; // inf 0, sup 1
   boolean Simu_Anneal = true; //  true ou false
   boolean Scal = true; //  true ou false
   String Scal_Type = "Position";  // "Position", "Lin" ou "Exp"
   int Nbr_Gener_GA = 1000; // inf 10
   int Aff_Level = 2; // inf 1, sup 5

   // Main
     Popu_Bin1_Struct P = new  Popu_Bin1_Struct ();
     P.Set_Param(Nbr_Neur, Nbr_In, Nbr_Out, iter_j, iter_BP,
fichier);
```

```
        P.Affichage_Level(Aff_Level);
        P.ModifNombrePop(Nbr_Indiv);
        P.ModifTauxCrois(Taux_Crois);
        P.ModifTauxMut(Taux_Mut);

        if(Scal == true)       P.Scaling(Scal_Type);
        if(Simu_Anneal == true)       P.Simulated_Annealing();
        if(Elitis == true) P.Elitisme_Set(Taux_Elitis);

        P.Initialisation();

        for(int i=0;i<Nbr_Gener_GA;i++) {
           P.Evaluation();
           P.Affiche();
           P.Selection();
           P.Croisement();
           P.Mutation();
         }
    }
}
```