MASTER EN SCIENCES MATHÉMATIQUES

Modélisation et apprentissage des réseaux de neurones artificiels

NICOLAY, Delphine

Award date:
2012

MASTER EN MATHÉMATIQUES

Modélisation et apprentissage des réseaux de neurones artificiels

Delphine Nicolay

2012

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR

Faculté des Sciences

MODELISATION ET APPRENTISSAGE DES RESEAUX DE NEURONES ARTIFICIELS

Mémoire présenté pour l'obtention

du grade académique de master en « sciences mathématiques »

Delphine NICOLAY

Juin 2012

Remerciements

Je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce mémoire.

Je tiens à remercier tout particulièrement Monsieur Carletti qui, en tant que promoteur de ce mémoire s'est toujours montré à l'écoute et très disponible tout au long de la réalisation de ce travail. Merci à lui pour son aide, sa patience et pour le temps qu'il m'a consacré.

Merci également à Irene Poli, Davide de March et Andrea Roli pour leur supervision lors de mon stage en Italie. Merci à eux pour leur accueil, leurs conseils et leur disponibilité.

Je tiens aussi à remercier Stéphanie Poncelet et Jeannine Bonne pour leur relecture attentive de ce mémoire.

Enfin, j'aimerais remercier mes parents, mes frères et ma soeur pour le soutien qu'ils m'ont apporté tout au long de mes études et particulièrement lors de l'élaboration de ce travail.

Merci à tous et à toutes.

Résumé

Les réseaux de neurones artificiels sont des modèles de calcul qui s'inspirent du fonctionnement des réseaux de neurones biologiques. Ils sont composés de neurones d'entrées, de neurones intermédiaires et de neurones de sorties reliés par des connexions. Ils reçoivent des signaux par leurs neurones d'entrées et produisent une réponse qu'ils renvoient à l'aide de leurs neurones de sorties. La réponse produite dépend des connexions qui existent entre les différents neurones. Ces réseaux sont capables d'apprendre à réaliser une tâche.

Les réseaux de neurones artificiels sont classés dans la famille des méthodes statistiques et des méthodes de l'intelligence artificielle. Ils sont utilisés dans des domaines variés tels que l'estimation boursière, la météorologie et l'approximation de fonctions complexes.

L'objectif de ce mémoire est d'étudier la modélisation et l'apprentissage de ces réseaux de neurones. La modélisation est étudiée de façon générale et à travers deux modèles particuliers. L'apprentissage est tout d'abord étudié à travers les algorithmes dits "classiques". Il est également relié aux algorithmes génétiques multi-objectifs. Une implémentation d'un modèle simple de réseau de neurones et de deux types d'algorithmes génétiques multi-objectifs est ensuite réalisée. Des tests sont effectués afin de trouver les paramètres qui permettent la convergence de ces algorithmes en un nombre minimal de générations. Cette première phase d'analyse est suivie d'une seconde phase au cours de laquelle la dégénérescence et la redondance des réseaux de neurones sont étudiées. Cette seconde phase permet de mettre en évidence les fonctions modélisables par le modèle de réseaux implémenté. Elle permet également une comparaison des résultats obtenus par optimisation modulaire et globale. Enfin, une application au domaine de la robotique évolutionnaire des réseaux de neurones et des algorithmes génétiques implémentés est expérimentée.

Mots-clés : réseaux de neurones artificiels, modélisation, apprentissage, algorithmes génétiques simples et multi-objectifs, robotique, optimisation.

Abstract

Artificial neural networks are calculation models inspired by the working of biological neural networks. They are made of input neurons, hidden neurons and output neurons linked by connections. They receive signals through their input neurons and they produce an answer that they send through their output neurons. The produced answer depends on connections existing between neurons of the network. Those networks are able to learn how to perform a particular task.

Artificial neural networks are classified as methods of statistics and artificial intelligence. They are used in various fields such as stock market valuation, meteorology and approximation of complex functions.

The main goal of this master's thesis is to study the modeling and the learning of artificial neural networks. The modeling is studied in a general way and through two particular models. First of all, the learning is studied through "classical" algorithms. It is also related to multi-objective genetic algorithms. Then, an implementation of a simple model of neural network and of two kinds of multi-objective genetic algorithms is performed. Some tests are run to find parameters which allow the convergence of algorithms in a minimum number of generations. This first phase of analysis is followed by a second one in which the degeneracy and the redundancy of neural networks are studied. This second phase allows to highlight functions which can be modeled by the kind of networks implemented. It also allows us to compare the results obtained with modular and global optimization. Finally, an application of implemented neural networks and genetic algorithms to the field of evolutionary robotics is tested.

Keywords : artificial neural networks, modeling, learning, simple and multi-objective genetic algorithms, robotics, optimization.

Table des matières

Introduction **1**

1 Fondements biologiques des modèles **3**
1.1 Architecture . . . 3
1.1.1 Structure du cerveau . . . 3
1.1.2 Structure des neurones . . . 5
1.2 Transmission de l'information . . . 6
1.2.1 Membrane cellulaire . . . 6
1.2.2 Création et déplacement du signal . . . 7
1.3 Utilisation et stockage de l'information . . . 9
1.4 Conclusion . . . 10

2 Modèles de réseaux de neurones artificiels **11**
2.1 Modèle général . . . 11
2.1.1 Modèle d'un neurone . . . 12
2.1.2 Modèle de réseaux de neurones . . . 13
2.2 Modèle de McCulloch-Pitts . . . 13
2.2.1 Description du neurone . . . 14
2.2.2 Illustration sur les fonctions booléennes . . . 15
2.2.3 Interprétation géométrique . . . 18
2.2.4 Réseaux de neurones de McCulloch-Pitts . . . 19
2.3 Modèle du perceptron . . . 20
2.3.1 Description du neurone . . . 21
2.3.2 Illustration sur les fonctions booléennes . . . 22
2.3.3 Interprétation géométrique . . . 23
2.3.4 Réseaux de neurones : le perceptron . . . 24
2.4 Classification des réseaux de neurones . . . 25
2.4.1 Réseaux équivalents . . . 26
2.4.2 Réseaux non équivalents . . . 28
2.5 Conclusion . . . 29

3 Apprentissage classique des réseaux de neurones **30**
3.1 Généralités . . . 30
3.2 Exemples d'algorithmes d'apprentissage . . . 31
3.2.1 Algorithme d'apprentissage du perceptron . . . 31
3.2.2 Algorithme d'apprentissage par compétition . . . 36
3.3 Conclusion . . . 40

4 Algorithmes génétiques et apprentissage des réseaux 41
4.1 Algorithmes génétiques simples . . . 41
4.1.1 Terminologie . . . 42
4.1.2 Fonctionnement et schéma général . . . 42
4.1.3 Sélection . . . 44
4.1.4 Reproduction . . . 46
4.1.5 Mutation . . . 47
4.2 Algorithmes génétiques multi-objectifs . . . 48
4.2.1 Dominance et optimalité de Pareto . . . 48
4.2.2 Approche à priori . . . 50
4.2.3 Approche à posteriori . . . 50
4.3 Apprentissage des réseaux . . . 58
4.3.1 Implémentation . . . 58
4.3.2 Exemple d'utilisation : la fonction *ou* . . . 62
4.4 Conclusion . . . 66

5 Analyse des paramètres et des réseaux optimaux 67
5.1 Analyse des paramètres . . . 67
5.1.1 Algorithme génétique à fitness pondérée . . . 69
5.1.2 Algorithme génétique multi-objectif NSGA . . . 80
5.1.3 Comparaison entre nos deux algorithmes . . . 85
5.2 Analyse des réseaux optimaux . . . 88
5.2.1 Implémentation des fonctions logiques . . . 89
5.2.2 Optimisation modulaire ou globale . . . 95
5.3 Conclusion . . . 98

6 Application à la robotique 99
6.1 Concept général . . . 99
6.2 Première expérience : suivre un gradient . . . 101
6.2.1 Modélisation du problème . . . 101
6.2.2 Résultats . . . 104
6.3 Deuxième expérience : éviter les obstacles . . . 107
6.3.1 Modélisation du problème . . . 107
6.3.2 Résultats . . . 109
6.4 Troisième expérience : suivre un gradient et éviter les obstacles . 111
6.4.1 Modélisation du problème . . . 111
6.4.2 Résultats . . . 113
6.5 Réflexion sur les résultats de nos expériences . . . 118
6.5.1 Résumé . . . 118
6.5.2 Remarques . . . 119
6.5.3 Perspectives . . . 119
6.6 Simulateur ARGoS . . . 120
6.6.1 Description . . . 120
6.6.2 Utilisation sur nos expériences . . . 122
6.7 Conclusion . . . 122

Conclusions et perspectives 123

Bibliographie - Annexes 125

Introduction

Le cerveau humain est une structure extrêmement complexe composée de plusieurs milliards de cellules, appelées neurones. Ces neurones sont connectés les uns aux autres et interagissent pour contrôler les réactions de notre corps en fonction des variations environnementales. Lorsqu'un changement se produit dans notre environnement, le cerveau reçoit les informations perçues par les sens et produit une réponse qui va conditionner notre comportement face à la situation. Le cerveau est capable d'apprendre. En effet, tout au long de notre vie, nous nous retrouvons face à de nouvelles situations et nous réagissons à celles-ci. Si notre réaction n'est pas adaptée, nous sommes capables de rectifier notre comportement de façon à ne plus commettre la même erreur.

Un réseau de neurones artificiels est un modèle de calcul qui s'inspire du fonctionnement des neurones biologiques. Comme ces derniers, ils reçoivent des informations et produisent une réponse en fonction des connexions qui les lient et des poids de ces connexions. Les réseaux de neurones artificiels sont classés dans la famille des méthodes statistiques et des méthodes de l'intelligence artificielle. En effet, ils permettent d'une part de réaliser des classifications et d'autre part ils fournissent des réponses indépendantes de l'utilisateur. Ils ont de bonnes capacités de classification et de généralisation. Comme les réseaux de neurones biologiques, les réseaux de neurones artificiels sont capables d'apprendre à réaliser une tâche. Ils sont utilisés dans des domaines aussi variés que l'estimation boursière, la météorologie (classification des conditions atmosphériques) et l'approximation de fonctions inconnues ou complexes.

L'objectif de ce mémoire est d'étudier la modélisation et l'apprentissage des réseaux de neurones artificiels. Nous aurons tout d'abord une phase d'étude théorique. Au cours de celle-ci, nous introduirons différents modéles de réseaux de neurones et nous présenterons deux méthodes d'apprentissage pour ces réseaux : l'apprentissage classique et l'apprentissage utilisant les algorithmes génétiques. Nous verrons que l'avantage de ce second apprentissage est qu'il permet de travailler à la fois sur les poids et l'architecture du réseau. Nous aurons ensuite une phase plus pratique durant laquelle nous implémenterons un modèle de réseaux neuronaux et des algorithmes génétiques. Le but de cette seconde phase sera d'analyser les réseaux optimaux obtenus à l'aide de nos algorithmes et d'étudier leur dégénérescence et leur redondance.

Nous commencerons ce mémoire par l'introduction des fondements biologiques des réseaux de neurones artificiels. Ces fondements comprennent la structure ainsi que les processus de transmission de l'information et d'apprentissage. Nous

en profiterons pour souligner les éléments des neurones biologiques qui devront impérativement se retrouver dans nos neurones artificiels.

Nous passerons ensuite à la modélisation de nos réseaux. Nous en présenterons tout d'abord une modélisation générale, c'est-à-dire leur forme générale et les éléments indispensables à cette modélisation. Nous étudierons ensuite deux modèles particuliers de réseaux : le modèle de McCulloch-Pitts et le modèle du perceptron. Nous illustrerons l'utilisation de ces deux modèles sur les fonctions booléennes. Nous terminerons par une classification des modèles de réseaux de neurones existants.

Le chapitre suivant sera consacré à l'apprentissage classique des réseaux de neurones, c'est-à-dire à l'étude des algorithmes créés pour optimiser les poids et les seuils de façon à ce que nos réseaux soient capables de réaliser une tâche particulière. Nous introduirons deux types d'apprentissage, à savoir l'apprentissage supervisé et non-supervisé. Nous donnerons un exemple d'algorithme pour les deux types d'apprentissage présentés.

Le chapitre 4 aura pour but de faire le lien entre l'apprentissage de nos réseaux et les algorithmes génétiques. Nous commencerons par présenter les algorithmes génétiques simples et multi-objectifs de façon théorique. Nous verrons ensuite pourquoi il est avantageux d'utiliser ces algorithmes lors du processus d'apprentissage de nos réseaux. Nous terminerons ce chapitre en donnant quelques indications sur la façon dont nous avons implémenté les réseaux de neurones et les algorithmes génétiques.

Nous aurons ensuite un chapitre consacré à deux phases d'analyse. La première phase servira à trouver les valeurs des paramètres de l'algorithme génétique qui permettent d'optimiser sa vitesse de convergence, c'est-à-dire qui permettent de minimiser le nombre de générations nécessaires à l'apparition du réseau optimal réalisant la tâche souhaitée. La seconde phase aura pour objectif d'analyser les réseaux optimaux obtenus lors de l'exécution de nos algorithmes. Nous verrons quelles sont les fonctions modélisables à l'aide du modèle de réseaux implémentés. Nous étudierons également la redondance et la dégénérescence de nos réseaux.

Le dernier chapitre consistera en une application de la modélisation et de l'apprentissage de nos réseaux au domaine de la robotique. Nous utiliserons nos réseaux pour contrôler des robots qui doivent réaliser une tâche particulière. La phase d'apprentissage de nos robots sera exécutée à l'aide des algorithmes génétiques implémentés. Les expériences réalisées au cours de ce chapitre seront des expériences abstraites. Nous terminerons par l'introduction du simulateur ARGoS et par une réflexion sur la façon de transformer nos tâches abstraites en tâches réalisables à l'aide de ce simulateur.

Chapitre 1

Fondements biologiques des modèles

Le but de ce chapitre est de comprendre les architectures et mécanismes biologiques qui ont inspiré les modèles que nous étudierons par la suite. Nous allons commencer par donner une brève description de l'architecture du cerveau jusqu'au niveau qui nous intéresse, c'est-à-dire jusqu'aux neurones. Nous allons également décrire le processus de transmission de l'information entre ceux-ci. Nous terminerons ce chapitre en donnant quelques éléments de réponse sur la façon dont ils mémorisent les informations reçues.

Ce chapitre est basé sur les références publiées par Gerstner et Kistler [14], Peretto [21] et Rojas [24].

1.1 Architecture

1.1.1 Structure du cerveau

« La structure générale du cerveau a une organisation remarquable qui n'est pas sans rappeler l'organisation d'un ordinateur central » (Peretto [21]). Nous allons détailler les principales structures qui composent le cerveau. La figure 1.1 est une représentation globale de celui-ci et indique la position de la plupart de ces structures principales.

Les principaux éléments qui composent le cerveau sont énoncés brièvement ci-dessous.

- Le cortex. C'est le processeur central du système.
- Le thalamus. C'est par lui que transitent toutes les informations qui arrivent et repartent du cortex.
- Les structures périthalamiques. Ce sont des structures qui ont des rôles auxilliaires. Il est à signaler que ces rôles ne sont pas encore complètement connus. Voici une brève description de ces différentes structures ainsi que des rôles qui leur sont alloués :
  - L'hypothalamus. Il contient des programmes fondamentaux pour notre survie et contrôle les sécrétions hormonales.

- La formation réticulée. Elle coordonne et exécute les programmes hypothalamiques.
- La formation nigrostriate. Cette structure est responsable de la coordination des actions de longue durée.
- Le cervelet. Il est impliqué dans le stockage, la récupération et l'ajustement précis de séquences de mouvements coordonnés.
- L'hippocampe. Cet élément joue un rôle essentiel dans le processus de stockage des informations dans la mémoire à long-terme.
- Le collicus. Il dirige les mouvements oculaires.

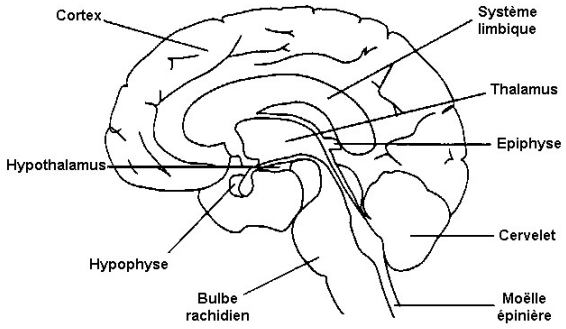

FIGURE 1.1 – Structure générale du cerveau et position de ces principaux éléments : cortex, thalamus et structures périthalamiques. Source : `http://tncorpshumain.tableau-noir.net`.

Nous pouvons à présent nous attarder quelques instants sur la structure du processeur central de notre cerveau, à savoir le cortex. Celui-ci est composé de cellules neuronales appelées neurones qui sont les éléments de base de l'architecture du cerveau. C'est une structure faite de deux hémisphères qui ont chacun une superficie de 11 dm^2. Ces deux hémisphères sont reliés par les 800 millions de fibres contenues dans le corps calleux. Ils sont divisés en plusieurs aires corticales (voir figure 1.2) qui ont des rôles différents. On peut ainsi parler des aires corticales visuelles, auditives et motrices.

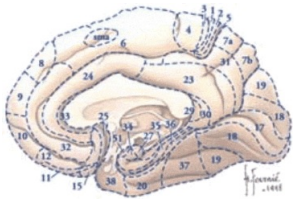

FIGURE 1.2 – Représentation des aires corticales du cerveau humain. Chacune de ces aires est composée de cellules neuronales et a un rôle particulier. On parle d'aires corticales visuelles, auditives et motrices. Source : `http://www.memoireonline.com/04/08/1036/prise-en-charge-infirmite-motrice-cerebrale-kinshasa.html`.

Il est possible d'affiner notre description en définissant des sous-ensembles dans les différentes aires corticales comme les bandes corticales mais ce n'est pas le but de ce travail. Nous allons passer à une description du neurone.

1.1.2 Structure des neurones

Comme nous l'avons vu précédemment, les neurones sont les cellules du cerveau. Dans cette partie, nous profiterons de la description biologique du neurone pour indiquer les éléments de celui-ci qui seront indispensables à la construction de nos modèles.

Les neurones reçoivent des signaux et produisent une réponse. Ils peuvent avoir des formes variables suivant leur fonction (neurones intermédiaires, neurones qui reçoivent les signaux envoyés par les sens,...). Toutefois, une architecture commune peut être reconnue et est représentée dans la figure 1.3.

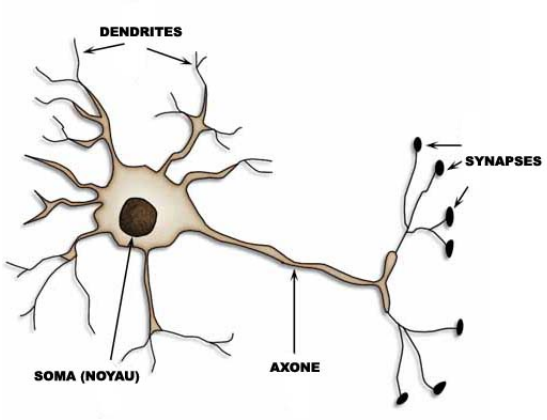

FIGURE 1.3 – Représentation d'un neurone et des principaux éléments qui le composent. Les dendrites reçoivent les signaux par les synapses et la cellule transmet un signal de sortie par l'axone. Le neurone, comme toute cellule, possède un noyau qui produit les substances chimiques nécessaires à sa subsistance. Source : `http://www.biotechnozen.com/articles/neurones.html`.

Les éléments principaux de ces cellules sont les dendrites, l'axone, les synapses et le corps de la cellule. Les dendrites reçoivent les signaux des régions de contact qu'elles possèdent avec les autres cellules. Ces régions de contact sont les synapses. Le signal de sortie (la réponse du neurone) est transmis par l'axone. Les organites du corps de la cellule produisent les substances chimiques nécessaires au bon fonctionnement de la cellule neuronale comme dans toute autre cellule.

Les quatre éléments énoncés ci-dessus devront être présents dans nos modèles pour que ceux-ci respectent la structure fondamentale du neurone. Nos neurones artificiels auront donc des canaux d'entrées, un corps et un canal de sortie. Les synapses seront représentés par des points de contact entre le corps et les connexions d'entrées ou de sortie.

1.2 Transmission de l'information

Les neurones transmettent l'information via des signaux électriques. Nous allons donner une approche de la façon dont ces signaux sont formés et transmis. Pour cela, nous devons tout d'abord introduire les propriétés et le comportement de la membrane cellulaire, ainsi que l'état du neurone au repos.

1.2.1 Membrane cellulaire

Commençons par rappeler la définition d'un ion (Larousse 2010 [1]) :

Définition 1.2.1 *Un ion est un atome ou un groupe d'atomes ayant gagné ou perdu un ou plusieurs électrons.*

Comme toutes les cellules, les neurones possèdent une membrane cellulaire. Celle-ci est composée de deux couches de molécules et forme une barrière de diffusion. Certains sels présents dans notre corps se dissolvent dans le fluide intra et extra-cellulaire et se dissocient en ions positifs et négatifs.

Les membranes des cellules possèdent une certaine perméabilité, c'est-à-dire un certain niveau d'échanges permis entre le fluide intra et extra-cellulaire, pour chacun des ions présents dans notre corps. Cette perméabilité est caractérisée par le nombre et la taille des pores, appelés canaux ioniques, par lesquels se font ces échanges. Les niveaux de perméabilité de la membrane sont différents d'un ion à l'autre, ce qui mène à une distribution différente des ions à l'intérieur et à l'extérieur de la cellule.

Le comportement de la membrane cellulaire peut être illustré par le modèle électrique expliqué ci-dessous. Ce modèle a été développé par Hodgkin et Huxley [7] et est représenté par la figure 1.4.

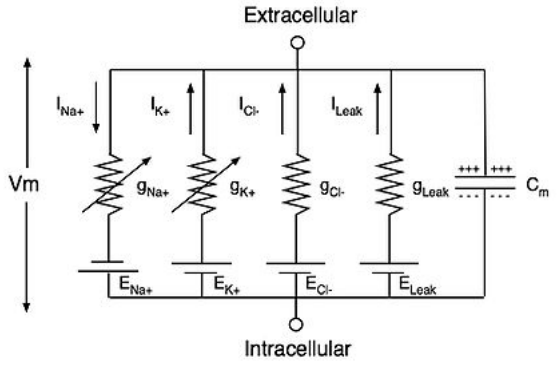

FIGURE 1.4 – Modèle électrique de la membrane cellulaire développé par Hodgkin et Huxley. La membrane cellulaire se comporte comme un condensateur. Une concentration différente des ions dans le fluide intra et extra-cellulaire fournit une source d'énergie capable de polariser l'intérieur de la cellule. Source : `http://en.wikipedia.org/wiki/Quantitative_models_of_the_action_potential`.

La membrane se comporte comme un condensateur fait de deux couches isolées de lipides. Les concentrations différentes des ions de part et d'autre de cette membrane, conséquence de sa perméabilité, fournissent une source d'énergie capable de polariser négativement l'intérieur de la cellule. Dans la figure 1.4, la perméabilité de la membrane est modélisée sous forme de conductance (la réciproque de résistance).

Lorsque le neurone est au repos, c'est-à-dire qu'il ne reçoit pas de signal, on dit que le modèle est à l'équilibre. Dans ce cas, la différence de potentiel est de -70 mV. Nous disposons à présent de toute l'information nécessaire pour comprendre la façon dont les signaux sont formés et transmis.

1.2.2 Création et déplacement du signal

A l'intérieur d'un neurone

Un signal est produit suite à la modification de la polarité de la cellule et à la réaction des canaux ioniques face à cette modification. En effet, quand la différence de potentiel entre l'intérieur et l'extérieur de la cellule dépasse un certain seuil, les canaux ioniques s'ouvrent ou se ferment pour permettre le retour à l'équilibre. Le déplacement des ions provoque une impulsion électrique qui va servir à créer et à transporter les signaux neuronaux à l'intérieur du neurone. Ces signaux qui se déplacent sont représentés comme des vagues de dépolarisation qui traversent les axones des cellules en se régénérant automatiquement. Ces vagues sont également appelées potentiels d'action (voir figure 1.5).

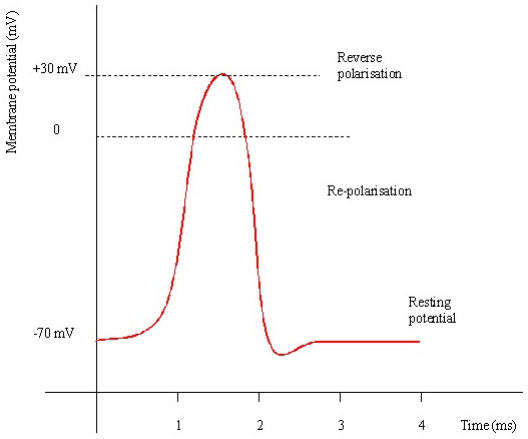

FIGURE 1.5 – Forme d'un potentiel d'action. A l'équilibre, la différence de potentiel entre l'intérieur et l'extérieur de la cellule est de -70 mV. Lorsqu'un signal est transmis, cette différence est modifiée. Dans cette figure, elle passe à $+30$ mV. Les canaux ioniques s'ouvrent alors pour permettre le retour à l'équilibre. Le déplacement des ions provoque une impulsion électrique qui permet de transmettre les signaux neuronaux à travers l'axone du neurone. Source : `http://www.antonine-education.co.uk/physics_a2/options/Module_6/Topic_4/topic_4__nerve_impulses.htm`.

En d'autres mots, une petite perturbation produite par le signal qui arrive des dendrites génère la polarisation de la membrane. Cette polarisation permet la création d'une vague de dépolarisation qui entraîne la transmission du signal le long de l'axone jusqu'au neurone suivant. Cette forme de transmission est connue sous le nom de modulation de fréquence. Elle permet de garder la précision du signal et demande peu d'énergie.

Entre les neurones

Nous venons de voir que le signal se déplace à travers le neurone par des potentiels d'action. Mais nous ne connaissons pas la cause du changement de polarité qui entraîne la création de ceux-ci. C'est ce que nous allons à présent décrire, ce qui nous permettra de comprendre la façon dont l'information passe d'un neurone à l'autre.

Nous avons vu que les neurones sont reliés par leurs synapses. Ceux-ci sont formés de petites vacuoles, les vésicules synaptiques, qui contiennent des émetteurs chimiques. Il y a un écart entre le synapse et la cellule à laquelle il est attaché. On parle de l'écart synaptique.

Quand une impulsion électrique touche le synapse, les émetteurs chimiques contenus dans les vésicules synaptiques sont libérés et entrent en contact avec les canaux ioniques de la cellule à laquelle ce synapse est lié. Il existe alors deux comportements possibles suivant la nature des émetteurs chimiques. Dans le premier cas, ils vont augmenter le potentiel à l'intérieur de la cellule et préparer un potentiel d'action. On dit que le synapse cause l'excitation de la cellule. Si, au contraire, le potentiel à l'intérieur de la cellule est diminué, la probabilité de créer un potentiel d'action décroît pendant un certain temps. Dans ce second cas, le synapse est inhibiteur.

Il est important de noter que les synapses déterminent une direction dans la transmission de l'information. Nos réseaux de neurones artificiels devront donc être représentés par des graphes orientés.

Seuil de dépolarisation

Nous avons vu que la transmission de l'information est régulée par un seuil qui doit être dépassé pour que le signal continue à se propager. Chaque neurone reçoit plusieurs signaux. Les potentiels électriques entrant dans une cellule vont donc être sommés pour obtenir le potentiel total et cette somme sera alors comparée au seuil d'excitation. Tous ces potentiels n'ont pas la même intensité et n'auront donc pas le même poids dans la somme. Nous devons noter que certains potentiels peuvent avoir un poids négatif (synapse inhibiteur).

Ces différents éléments devront se retrouver dans notre modèle pour garder un certain réalisme. Chaque neurone artificiel aura donc un seuil qui devra être atteint pour que la cellule soit excitée. Les différents canaux d'entrées d'un neurone auront éventuellement des poids différents et seront sommés (ou déduits selon leur signe). Le total sera ensuite comparé au seuil d'excitation. La réponse du neurone dépendra du résultat de cette comparaison.

1.3 Utilisation et stockage de l'information

Les informations fournies par les sens et transmises aux cellules neuronales vont permettre l'interaction entre l'individu et son environnement. Grâce à ces informations, l'individu peut apprendre à réagir face aux différentes situations qu'il rencontre. Il peut également retenir certaines de ces informations afin d'être capable de réagir rapidement et correctement lorsqu'il se retrouve face à une situation déjà rencontrée. On dit que l'individu peut apprendre et mémoriser.

L'apprentissage et la mémorisation sont permis par la plasticité cérébrale. Ce terme désigne la capacité de malléabilité du cerveau en fonction de son environnement. Cette plasticité se manifeste de deux façons différentes. La première consiste en l'apparition et la disparition de synapses, ce qui entraîne des changements de connexions entre les neurones. La seconde se présente sous forme de plasticité d'un neurone ou d'un synapse même, ce qui signifie que les seuils d'excitation des neurones et les poids des synapses peuvent être modifiés en fonction de l'environnement de l'individu.

Lors de l'apprentissage, les deux types de plasticité sont utilisées. En effet, les connexions inutiles existant entre certains neurones disparaissent. De plus, les neurones sont capables de modifier leur seuil d'excitation pour s'adapter à de nouvelles situations. Le poids de chaque synapse peut également être modifié lors de ce processus d'apprentissage.

Dans les réseaux de neurones, les informations sont stockées et mémorisées grâce à la plasticité synaptique. Celle-ci ne consiste pas simplement en la modification du poids du synapse. En effet, elle repose également sur le principe que l'efficacité de la transmission synaptique entre deux neurones connectés l'un à l'autre est fonction de l'activité passée de ces neurones (règle de Hebb). Un tel mécanisme aide à stabiliser l'apprentissage de certains comportements, c'est-à-dire à mémoriser la réponse à donner en fonction de la situation.

Il existe deux formes principales de plasticité synaptique : la potentialisation à long terme (LTP) et la dépression à long terme (LTD). La LTP est un mécanisme qui permet le renforcement durable des synapses entre deux neurones qui sont activés simultanément. Ce renforcement entraîne une réponse facilitée à une stimulation future. La LTD est le phénomène inverse. Il conduit à la mise sous silence des synapses qui interviennent dans des réseaux de neurones impliqués dans des mouvements erronés. Ce mécanisme est important pour la mémorisation des habitudes, des comportements intuitifs. Ces deux mécanismes sont dirigés par des processus chimiques intervenant lors de la transmission de l'information.

Dans ce travail, nous nous intéresserons à l'apprentissage de nos réseaux de neurones. Nous pourrons alors modifier la structure de nos réseaux, ainsi que les valeurs des poids des synapses et des seuils d'excitation. Bien que certains modèles de réseaux de neurones soient construits en tenant compte de la règle de Hebb, nous ne nous en occuperons pas par la suite.

1.4 Conclusion

Dans ce chapitre, nous avons introduit les fondements biologiques qui ont inspiré les modèles de réseaux de neurones que nous étudierons dans le chapitre suivant. Nous avons souligné les éléments importants de la structure du neurone, ainsi que ceux intervenant lors de la transmission et du stockage de l'information. Tous ces éléments seront présents dans nos modèles. Nous terminons cette conclusion par deux remarques concernant le lien entre les réseaux de neurones biologiques et les réseaux de neurones artificiels.

Au cours de notre vie, les neurones de notre cerveau commencent à mourir dès que nous atteignons notre majorité et ne sont pas remplacés. Il est donc plus que probable que chaque tâche soit assignée à une zone neuronale plutôt qu'à un seul neurone. Dans ce cas, nous pourrions imaginer que chaque neurone de nos réseaux de neurones artificiels représente une de ces zones et non pas un seul neurone biologique. Par exemple, le neurone lié à l'entrée représente la zone neuronale qui reçoit les signaux envoyés par l'un de nos cinq sens. Cette façon de considérer nos réseaux est très intéressante. En effet, lors de nos analyses nous travaillerons sur des réseaux de petite taille et cette représentation de nos réseaux sera alors plus cohérente avec l'inspiration biologique étant donné les millions de connexions existant dans le cerveau.

Bien que chacun de nos neurones artificiels puissent être considérés comme une zone neuronale et, qu'à l'origine, les modèles de réseaux de neurones aient été développés dans l'optique de représenter les réseaux de neurones biologiques, il faut être conscient que nos modèles restent très éloignés de la réalité. En effet, la structure des neurones artificiels est très simplifiée par rapport à celle des neurones biologiques. De plus, les connexions à l'intérieur du cerveau sont extrêmement complexes et le rôle de certains de ces composants n'est pas encore élucidé.

Chapitre 2

Modèles de réseaux de neurones artificiels

Dans le premier chapitre, nous avons introduit les éléments fondamentaux qui composent un neurone ainsi qu'un réseau de neurones. Nous allons à présent pouvoir modéliser ceux-ci. Nous allons tout d'abord donner une description générale de la façon de modéliser un neurone avant de passer à la modélisation d'un réseau de neurones. Nous décrirons ensuite des modèles particuliers souvent rencontrés dans la littérature et, pour chacun d'eux, nous donnerons des exemples de ce qu'ils peuvent réaliser comme tâche en travaillant sur les fonctions booléennes. Nous terminerons ce chapitre par une classification des réseaux de neurones en fonction de leurs caractéristiques.

A partir de ce chapitre, nous allons adopter une terminologie précise. Nous allons parler de neurones pour les neurones artificiels. Si nous parlons du neurone biologique, nous le préciserons.

Ce chapitre est inspiré de MacKay [19], Rojas [24] et Younes [30]. Toutes les figures présentées pour illustrer les concepts introduits proviennent également de ces deux références.

2.1 Modèle général

Nous avons vu dans le chapitre précédent que les neurones biologiques sont composés de quatre éléments principaux :

- les dendrites,
- le corps de la cellule,
- l'axone,
- les synapses.

Comme nous l'avions dit, nos neurones seront composés de canaux d'entrées, d'un corps, d'un canal de sortie et de points de connexions entre le corps du neurone et les différents canaux qui y sont reliés.

Nous avons également vu que chaque neurone biologique possède un seuil qui doit être dépassé pour que la cellule soit excitée et pour qu'elle transmette l'information. Nous allons devoir tenir compte de cette caractéristique dans nos modèles. Nous ne devons pas non plus oublier que nos réseaux devront être représentés par des graphes orientés pour respecter le flux directionnel de l'information.

2.1.1 Modèle d'un neurone

La figure 2.1 est une représentation générale du neurone. Celui-ci possède n canaux d'entrées $x_1, x_2, \ldots, x_n$. Sa sortie est une fonction qui dépend de ces n entrées.

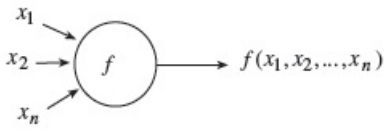

FIGURE 2.1 – Représentation du modèle général d'un neurone. Il possède n entrées et 1 sortie. Cette dernière est définie comme fonction des n entrées.

En général, nous préférerons utiliser des fonctions très simples dans les évaluations de sortie du neurone. Nous utiliserons donc des fonctions qui ne demandent qu'un argument d'entrée. Nous devrons alors commencer par réduire les n entrées à une seule avant d'évaluer la fonction de sortie. Pour ce faire, la fonction la plus souvent utilisée est l'addition. Dans la figure 2.2, la fonction de sortie est f et celle d'addition est g.

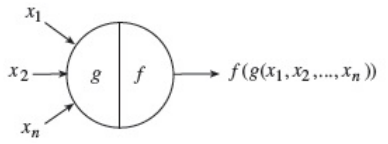

FIGURE 2.2 – Représentation du modèle général d'un neurone dans le cas où la fonction de sortie est une fonction à un argument. Elle est représentée par la lettre f dans cette figure. La lettre g, quant à elle, représente la fonction qui rassemble les n entrées en un argument avant d'utiliser la fonction de sortie. La plupart du temps, g est la fonction d'addition.

Remarque
Les entrées ont parfois des poids différents qui reproduisent les différentes intensités des signaux perçus. Dans ce cas, ces poids sont représentés par des valeurs situées au milieu de chacune des connexions. Une connexion sans valeur correspond à un poids unitaire.

A présent que nous avons modélisé un neurone, nous allons pouvoir nous attaquer à la modélisation d'un réseau de neurones.

2.1.2 Modèle de réseaux de neurones

Les réseaux de neurones biologiques nous permettent d'accomplir différentes actions. Par exemple, à partir des informations fournies par nos yeux, nous allons pouvoir situer un objet et le prendre en main. Chaque action peut être représentée par une fonction. Un réseau de neurones peut donc être vu comme le modèle d'une fonction $F : \mathbf{R}^n \to \mathbf{R}^m$ où n indique le nombre d'entrées et m le nombre de sorties. Le réseau est construit à partir de plusieurs neurones qui sont reliés les uns aux autres. Les connexions entre ces différents neurones sont orientées.

La figure 2.3 est un exemple de réseau de neurones. La fonction représentée est une fonction $F : \mathbf{R}^3 \to \mathbf{R}$ qui est composée de 4 neurones. On peut voir qu'il y a des connexions orientées entre ces différents neurones. Les canaux d'entrées n'ont pas de poids mais les connexions internes entre les différents neurones en ont.

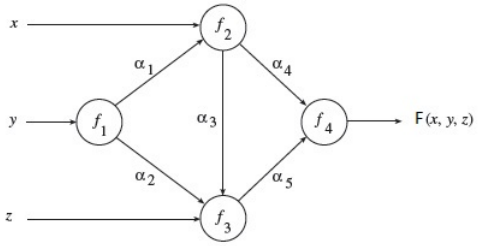

FIGURE 2.3 – Exemple de réseau de neurones. Il est composé de 4 neurones et modélise une fonction $F : \mathbf{R}^3 \to \mathbf{R}$. Ses canaux d'entrées n'ont pas de poids tandis que ses connexions internes en sont pourvues.

Remarque
Certaines fonctions simples pourront être modélisées à l'aide d'un seul neurone.

Il existe un grand nombre de modèles de réseaux de neurones. Nous avons décidé d'en présenter deux dans la suite de ce chapitre. Nous commencerons par présenter le modèle de McCulloch-Pitts. Nous présenterons ensuite le modèle du perceptron. Ces deux modèles possèdent des caractéristiques assez différentes qui nous permettront de faire une classification élémentaire des différents réseaux de neurones existants.

2.2 Modèle de McCulloch-Pitts

Le modèle de McCulloch-Pitts est le premier modèle qui a été développé. Il date de 1943 et a été proposé par Warren McCulloch et Walter Pitts. C'est un modèle assez ancien mais il va nous permettre de comprendre les bases d'un modèle et de faire une première analyse des tâches réalisables pour un réseau de neurones de ce type.

Premièrement, nous allons décrire un neurone construit sur ce modèle et les

règles qui régissent la valeur de sa sortie. Nous illustrerons ensuite l'utilisation de ce type de neurones par la modélisation des fonctions booléennes. Nous donnerons également une interprétation géométrique des fonctions qui peuvent être modélisées par un seul neurone. Nous terminerons par une analyse des fonctions modélisées par nos réseaux de neurones.

2.2.1 Description du neurone

Les neurones de McCulloch-Pitts utilisent des signaux binaires, ce qui signifie que les entrées du neurone et sa sortie peuvent prendre deux valeurs particulières : 0 et 1. Les arêtes n'ont pas de poids et peuvent être excitatrices ou inhibitrices. Pour les différencier, nous plaçons un petit rond vide à la fin des arêtes inhibitrices. Chaque neurone a un certain seuil d'excitation noté θ. Un neurone de McCulloch-Pitts est représenté dans la figure 2.4. Nous pouvons voir que celui-ci a des arêtes excitatrices (x_1 par exemple) et une arête inhibitrice (x_2).

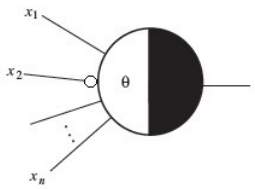

FIGURE 2.4 – Neurone de McCulloch-Pitts. Ce type de neurones utilise des signaux binaires. Ses arêtes n'ont pas de poids et peuvent être excitatrices (arêtes normales) ou inhibitrices (arêtes finies par une petit rond vide). Le seuil d'excitation du neurone est représenté par θ.

Supposons qu'un neurone de McCulloch-Pitts a n arêtes d'entrées $x_1, x_2, \ldots, x_n$ excitatrices et m arêtes d'entrées $y_1, y_2, \ldots, y_m$ inhibitrices. Les règles pour évaluer la sortie de ce neurone sont les suivantes :

- si $m \geq 1$ et si au moins un des signaux $y_1, y_2, \ldots, y_m$ est 1, le neurone est inhibé et sa sortie vaut 0.
- sinon, l'excitation totale $x = x_1 + x_2 + \cdots + x_n$ est calculée et comparée au seuil d'excitation θ. Si $x \geq \theta$, la sortie du neurone est 1. Dans le cas contraire, la sortie vaut 0.

Ces règles impliquent que le neurone peut être rendu inactif par un seul signal inhibiteur, ce qui est également le cas pour certains neurones biologiques. S'il n'y a pas de signaux inhibiteurs, nous avons un seuil à dépasser pour que le neurone soit excité.

La fonction d'activation du neurone de McCulloch-Pitts correspond à la fonction de sortie introduite dans le modèle général. Comme son nom l'indique, elle détermine la sortie du neurone. C'est une fonction en escalier. Elle passe de façon discontinue de 0 à 1 en θ. Si θ est nul, la fonction donnera toujours 1. Si θ est plus grand que n, nous aurons toujours une sortie nulle. La fonction d'activation est représentée dans la figure 2.5.

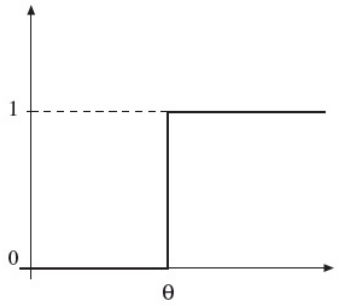

FIGURE 2.5 – Fonction d'activation (de sortie) d'un neurone de McCulloch-Pitts. C'est une fonction en escalier. Elle passe de façon discontinue de 0 à 1 en θ, qui est le seuil d'excitation du neurone.

2.2.2 Illustration sur les fonctions booléennes

Les fonctions booléennes, aussi appelées fonctions logiques, sont des fonctions qui donnent 1 ou 0 comme solution. La valeur de sortie 1 est associée à la valeur logique *vrai* tandis que la sortie 0 est associée à *faux*. Ces fonctions permettent de représenter des connecteurs logiques tels que *et, ou,...* ou des combinaisons de ceux-ci.

Nous allons illustrer le modèle de neurones de McCulloch-Pitts en analysant la façon dont nous pouvons modéliser ces fonctions particulières grâce à ce modèle.

Fonctions booléennes simples

Nous allons tout d'abord nous concentrer sur des fonctions logiques simples, à savoir *et* et *ou*. Nous allons supposer que nous avons deux entrées. Il nous suffira de généraliser pour connaître le neurone dans le cas avec n entrées. Les neurones qui réalisent ces deux fonctions sont représentés dans la figure 2.6.

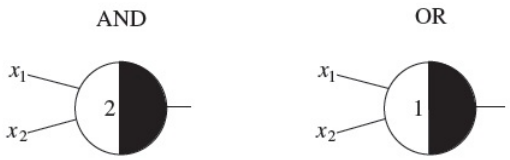

FIGURE 2.6 – Modélisation des fonctions logiques simples *et* et *ou* par des neurones de McCulloch-Pitts. Le cas considéré est celui en deux dimensions.

Nous allons vérifier que nous obtenons bien les tables de vérité attendues. Comme nous n'avons pas d'entrées inhibitrices, il nous suffit d'additionner les entrées et de comparer le résultat au seuil pour obtenir la valeur de sortie. La table 2.1 représente les sorties obtenues en fonctions des différentes combinaisons d'entrées pour les deux neurones de la figure 2.6.

x_1	x_2	Sortie
0	0	0
0	1	0
1	0	0
1	1	1

et

x_1	x_2	Sortie
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 2.1 – Sorties obtenues en fonction des différentes combinaisons d'entrées possibles pour les neurones de la figure 2.6. Le tableau de gauche correspond à la table de vérité de la fonction *et* et le tableau de droite à celle de la fonction *ou*. Les neurones de la figure 2.6 modélisent donc les fonctions voulues.

Ces deux tableaux de résultats correspondent bien aux tables de vérité des fonctions logiques *et* et *ou* respectivement. Les réseaux de la figure 2.6 modélisent donc ses fonctions voulues.

Importance des arêtes inhibitrices

Nous avons pu remarquer que ces deux fonctions ne nécessitent pas d'entrées inhibitrices pour être modélisées. Nous allons à présent montrer que cela n'est pas vrai pour toutes les fonctions logiques. Pour cela, nous allons tout d'abord introduire la définition de fonction logique monotone.

Définition 2.2.1 *Une fonction logique monotone de n arguments est une fonction dont les valeurs de deux points $x = (x_1, \ldots, x_n)$ et $y = (y_1, \ldots, y_n)$ sont telles que $f(x) \geq f(y)$ quand le nombre de 1 dans y est plus petit que dans x.*

Par exemple, la négation logique est une fonction logique à un argument qui est non-monotone. En effet, si $x = 1$ et $y = 0$, nous avons que $f(x) = 0$ et $f(y) = 1$. L'argument y a une valeur de fonction plus grande que x avec moins de 1.

Nous pouvons à présent énoncer la proposition suivante :

Proposition 2.2.1 *Les neurones de McCulloch-Pitts qui ne contiennent pas d'entrées inhibitrices peuvent seulement modéliser des fonctions logiques monotones.*

Preuve
Si nous n'avons pas d'entrées inhibitrices, nous devons comparer le seuil d'excitation à la somme des entrées. Supposons que x et y sont des vecteurs à n composantes et que l'ensemble des 1 de y est plus petit que celui de x. Nous allons analyser les différents cas qui peuvent se présenter :

- si $f(x) = 0$, alors $f(y) = 0$ car si la somme des composantes non-nulles de x n'atteint pas le seuil, la somme de celles de y, moins nombreuses, ne peut pas l'atteindre. Nous avons donc $f(x) \geq f(y)$.
- si $f(x) = 1$, c'est-à-dire si la somme des composantes de x atteint le seuil, nous avons deux possiblités :
  - soit la somme des composantes de y atteint également le seuil et $f(y) = 1$.
  - soit elle ne l'atteint pas et $f(y) = 0$.

  Au final, nous obtenons toujours $f(x) \geq f(y)$

Tous les cas possibles respectent la monotonie de la fonction logique.

La propriété d'inhibition de certaines arêtes est donc très importante. En effet, nous ne pourrions pas modéliser toutes les fonctions logiques sans celle-ci. Pour illustrer l'utilisation de ces arêtes inhibitrices, nous allons modéliser la négation logique que nous avions utilisée comme exemple de fonction logique non-monotone. Le neurone modélisant cette fonction est représenté dans la figure 2.7.

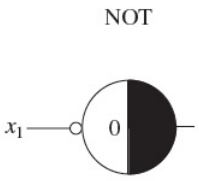

FIGURE 2.7 – Modélisation de la fonction logique non-monotone *not* par un neurone de McCulloch-Pitts. La propriété d'inhibition est indispensable pour pouvoir modéliser cette fonction.

Cette modélisation nous donne la table de sorties 2.2 qui correspond bien à la table de vérité de la négation logique.

x_1	Sortie
0	1
1	0

TABLE 2.2 – Table de sorties du neurone représenté dans la figure 2.7 en fonction des différentes entrées possibles. Cette table correspond bien à la table de vérité de la négation logique. Nous obtenons donc la modélisation désirée.

Combinaison de fonctions logiques

Pour terminer notre illustration, nous allons à présent modéliser deux fonctions logiques qui sont des combinaisons de celles qui ont été modélisées précédemment. Ces deux fonctions sont les fonctions logiques $x_1 \wedge \neg x_2$ et $\neg x_1 \wedge \neg x_2$. Leur représentation est donnée par la figure 2.8.

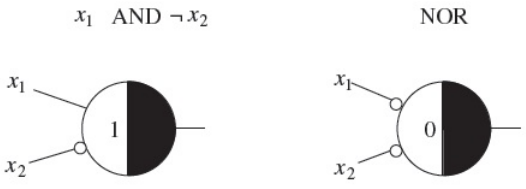

FIGURE 2.8 – Modélisation de combinaisons de fonctions logiques, à savoir $x_1 \wedge \neg x_2$ et *nor*, par des neurones de McCulloch-Pitts.

Les tables de sorties correspondant aux neurones de la figure 2.8 sont représentées dans la table 2.3. Ces deux tables correspondent bien aux tables de vérité des fonctions booléennes que nous voulions modéliser.

x_1	x_2	$\neg x_2$	Sortie
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

et

x_1	x_2	$\neg x_1$	$\neg x_2$	Sortie
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

TABLE 2.3 – Sorties obtenues en fonction des différentes combinaisons d'entrées pour les neurones représentés dans la figure 2.8. Ces sorties sont celles attendues pour les fonctions booléennes $x_1 \wedge \neg x_2$ et *nor*.

2.2.3 Interprétation géométrique

Nous allons à présent visualiser le genre de fonctions qui peuvent être représentées par les neurones de McCulloch-Pitts. Nous allons supposer que nous travaillons dans l'espace d'entrées à deux dimensions. Nous disposons alors de deux variables x_1 et x_2 qui peuvent chacune prendre les valeurs 0 ou 1. Nous aurons donc quatre combinaisons possibles. Chacun de ces quatre points va correspondre à un sommet du carré unitaire dans cet espace (voir figure 2.9).

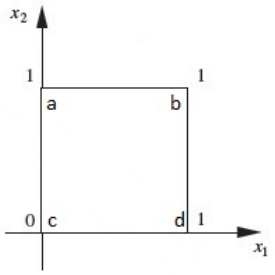

FIGURE 2.9 – Représentation géométrique des fonctions logiques à deux entrées. Chacune d'entre-elles peut prendre deux valeurs et il y a donc quatre combinaisons possibles. Celles-ci correspondent aux quatre sommets du carré unitaire.

La table 2.4 représente la position du point dans l'espace d'entrées en fonction des valeurs de x_1 et x_2. La position du point sur l'axe horizontal est donné par la valeur de x_1 et celle sur l'axe vertical par la valeur de x_2.

Les neurones de McCulloch-Pitts divisent cet espace d'entrées en deux sous-espaces. En effet, pour un vecteur d'entrées donné (x_1, x_2) et pour un seuil θ défini, nous testons la condition $x_1 + x_2 \geq \theta$. Cette condition est vérifiée pour tous les points d'un côté de la droite d'équation $x_1 + x_2 = \theta$ et est rejetée pour les points situés de l'autre côté.

x_1	x_2	Sommet
0	0	c
0	1	a
1	0	d
1	1	b

TABLE 2.4 – Position du point dans l'espace des entrées représenté par la figure 2.9. La position sur l'axe horizontal dépend de la valeur de la première entrée (x_1) et celle sur l'axe vertical de la seconde (x_2).

La figure 2.10 illustre cette séparation dans le cas où la condition testée est $x_1 + x_2 \geq \theta$ avec $\theta = 1$, qui est la condition de la fonction logique *ou*. Les points situés au-dessus de la droite donneront une sortie de 1 tandis que ceux situés en dessous auront une sortie égale à 0.

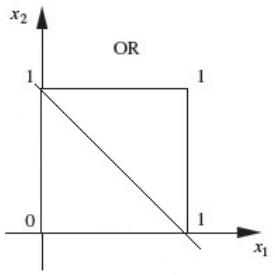

FIGURE 2.10 – Séparation de l'espace d'entrées de la fonction logique *ou*. Les points situés au-dessus de la droite donne une sortie de 1. Au contraire, ceux situés sous celle-ci ont une sortie égale à 0.

2.2.4 Réseaux de neurones de McCulloch-Pitts

Jusqu'à présent, nous nous sommes concentrés sur le modèle d'un neurone de McCulloch-Pitts et sur les fonctions logiques qui pouvaient être représentées par celui-ci. Cependant certaines fonctions logiques nécessitent l'utilisation d'un réseau de neurones pour pouvoir être modélisées. Nous allons essayer de voir si toutes les fonctions logiques peuvent être représentées par nos réseaux.

Nous allons commencer par introduire une nouvelle notion qui est la notion de décodeur.

Définition 2.2.2 *Un neurone est appelé décodeur d'un vecteur d'entrées donné s'il ne donne la valeur de sortie* 1 *que pour ce vecteur.*

Par exemple, le neurone qui modélise la condition $x_1 \wedge \neg x_2 \wedge x_3$ est un décodeur pour le vecteur d'entrées $(1, 0, 1)$ car il donne une sortie de 1 uniquement pour ce vecteur.

A partir de cette nouvelle notion, nous allons pouvoir montrer que n'importe

quelle fonction logique peut être modélisée par un réseau de neurones de McCulloch-Pitts. Pour cela, commençons par donner un exemple. Supposons que nous voulons modéliser la fonction F représentée par la table de sorties 2.5.

Vecteurs d'entrée	F
$(0,0,1)$	1
$(0,1,0)$	1
Tous les autres	0

TABLE 2.5 – Table de sorties qui représente la fonction F. Il est possible de modéliser cette fonction par un réseau de neurones de McCulloch-Pitts. Pour cela, nous utilisons des décodeurs.

Nous allons commencer par trouver un décodeur pour chacune des vecteurs d'entrées qui doivent donner un 1 en sortie. Ensuite, nous les relierons à un neurone implémentant la fonction logique *ou*. De cette façon, si l'un des deux vecteurs d'entrées donne un 1, nous aurons bien une sortie de 1 à la fin du réseau. Ce réseau est formé de deux couches de neurones. La première est composée des décodeurs tandis que la seconde correspond au neurone qui modélise la fonction *ou*. Le réseau obtenu est représenté dans la figure 2.11.

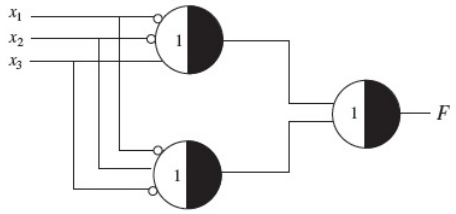

FIGURE 2.11 – Modélisation de la fonction logique F par un réseau de neurones de McCulloch-Pitts. Ce réseau est formé de deux couches. La première est composée de décodeurs, c'est-à-dire de neurones qui ne donnent la valeur de sortie 1 que pour un vecteur d'entrées particulier. La seconde modélise la fonction logique *ou*.

Il est évident que le raisonnement que nous venons de suivre pour modéliser la fonction F peut servir à modéliser n'importe quelle fonction logique. Nous pouvons donc énoncer une proposition sur la possibilité de modéliser n'importe quelle fonction logique à l'aide d'un réseau de neurones de McCulloch-Pitts.

Proposition 2.2.2 *N'importe quelle fonction logique* $F : \{0,1\}^n \to \{0,1\}$ *peut être modélisée par un réseau de neurones de McCulloch-Pitts à deux couches.*

Nous pouvons voir que même si le modèle de McCulloch-Pitts semble réducteur du fait de ces entrées binaires, il permet de modéliser un grand nombre de fonctions. Passons à présent à la description d'un second modèle : le perceptron.

2.3 Modèle du perceptron

Ce modèle a été proposé pour la première fois en 1958 par Frank Rosenblatt, un psychologue américain. Nous allons suivre le même schéma que pour le dé-

veloppement du modèle de McCulloch-Pitts.

2.3.1 Description du neurone

Les neurones qui composent le perceptron ont des caractéristiques différentes de ceux utilisés dans les réseaux de neurones de McCulloch-Pitts. Les entrées de ces neurones ne sont plus des valeurs binaires mais réelles. Les sorties, quant à elles, gardent des valeurs binaires. Dans ce modèle, les arêtes ont des poids différents. Ces poids peuvent être positifs ou négatifs. Si le poids est positif, l'arête est excitatrice. Sinon, elle est inhibitrice. La règle d'évaluation de ce type de neurone est semblable à celle de McCulloch-Pitts dans le cas sans arêtes inhibitrices. Une représentation d'un tel neurone est donnée par la figure 2.12. On l'appelle perceptron simple et sa définition est donnée ci-dessous.

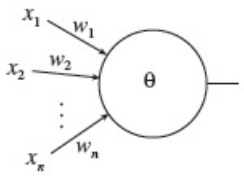

FIGURE 2.12 – Neurone qui compose le perceptron. Contrairement aux neurones de McCulloch-Pitts, ce neurone possède des poids sur ses arêtes. Toutes les entrées n'ont donc plus la même importance dans la somme. De plus, les entrées ne sont plus binaires mais réelles. Les sorties, par contre, restent binaires.

Définition 2.3.1 *Un perceptron simple est un neurone avec un seuil θ qui, quand il reçoit les n entrées réelles $x_1, x_2, \ldots, x_n$ à travers ses arêtes avec les poids associés $w_1, w_2, \ldots, w_n$, sort 1 si l'inégalité $\sum_{i=1}^{n} w_i x_i \geq \theta$ est vérifiée et sort 0 sinon.*

Il est parfois utile de travailler avec un seuil nul. Pour cela, il est possible de modifier légèrement notre neurone comme illustré par la figure 2.13. Le seuil de notre perceptron simple (à gauche) a été transformé en poids $-\theta$ d'une nouvelle entrée de valeur 1. Ce poids en plus est appelé le biais du neurone. Dans ce cas, le vecteur d'entrées passe de n composantes à $n+1$ et est alors appelé le vecteur d'entrées étendu. Le vecteur de poids doit également être étendu à $n+1$ composantes $(w_1, w_2, \ldots, w_n, w_{n+1})$ où $w_{n+1} = -\theta$.

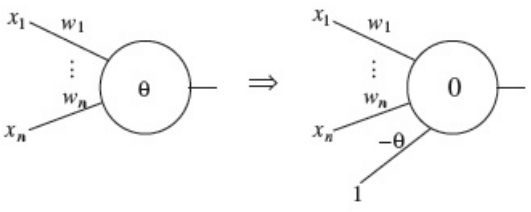

FIGURE 2.13 – Perceptron simple avec biais. Il est équivalent au perceptron simple mais possède une entrée supplémentaire de valeur 1 et de poids $-\theta$, ce qui lui permet d'avoir un seuil nul. Ce nouveau poids est appelé le biais du neurone.

2.3.2 Illustration sur les fonctions booléennes

Nous allons de nouveau nous intéresser aux fonctions logiques et nous allons tenter de répondre à la question suivante : Est-il possible de modéliser toutes les fonctions logiques à partir d'un perceptron simple ?

Pour avoir une première intuition de la réponse, nous allons tout d'abord nous intéresser aux fonctions logiques à deux arguments. Il est possible de créer 16 fonctions à partir de deux entrées en prenant toutes les combinaisons de sorties possibles. Ces 16 fonctions sont reprises dans la table 2.6.

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

TABLE 2.6 – Tables de vérité des 16 fonctions booléennes à deux arguments.

Nous allons montrer que la fonction f_6, qui correspond à la fonction *xor*, ne peut pas être modélisée par un perceptron simple. En effet, supposons que w_1 et w_2 soient les poids des entrées de ce perceptron et que θ soit son seuil. Si nous voulons que ce perceptron modélise la fonction *xor*, les quatres conditions du tableau 2.7 doivent être vérifiées.

x_1	x_2	$w_1x_1 + w_2x_2$	Sortie attendue	Condition
0	0	0	0	$0 < \theta$
0	1	w_2	1	$w_2 \geq \theta$
1	0	w_1	1	$w_1 \geq \theta$
1	1	$w_1 + w_2$	0	$w_1 + w_2 < \theta$

TABLE 2.7 – Tableau reprenant les sorties attendues en fonction des différentes combinaisons d'entrées et les conditions à satisfaire pour obtenir ces sorties. Ces conditions sont contradictoires.

De la première ligne du tableau, nous pouvons déduire que θ est positif, ce qui veut dire que w_1 et w_2 vont également être positifs par les lignes 2 et 3. De plus, chacun des deux est plus grand que θ. Leur somme ne peut donc pas donner un nombre plus petit que le seuil dans la condition de la ligne 4.

Cette contradiction implique qu'un perceptron simple ne peut modéliser la fonction *xor*. Cette fonction n'est pas la seule à ne pas pouvoir être modélisée par ce type de neurones. Nous pouvons citer la fonction f_9 comme autre fonction qui n'est pas modélisable. Celle-ci correspond à la vérification de l'identité des deux entrées (donne 1 si $x_1 = x_2$).

Nous allons tenter d'interpréter ces fonctions géométriquement pour voir si elles ont une propriété en commun qui permettrait d'expliquer pourquoi elles ne sont pas modélisables par un perceptron simple.

2.3.3 Interprétation géométrique

Comme lors de l'analyse des neurones de McCulloch-Pitts, nous allons travailler en deux dimensions. L'espace d'entrées est de nouveau le carré unitaire.

L'interprétation géométrique des fonctions est la même que pour le modèle de McCulloch-Pitts. En effet, un perceptron simple sépare l'espace d'entrées en deux sous-espaces. Les points qui appartiennent à un des sous-espaces auront une sortie égale à 1 tandis que ceux appartenant à l'autre vaudront 0. Contrairement au modèle précédent, la droite qui sépare les deux sous-espaces ne passe pas forcément par un des sommets du carré unitaire. La figure 2.14 est une représentation de la séparation d'un espace d'entrées par l'équation $0.9x_1 + 2x_2 \geq 1$.

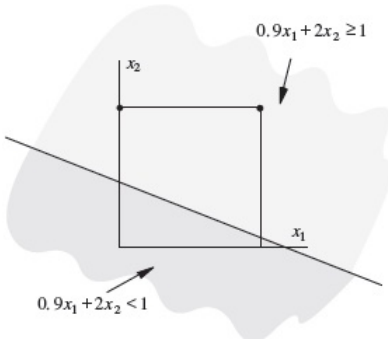

FIGURE 2.14 – Séparation de l'espace d'entrées par l'équation $0.9x_1 + 2x_2 \geq 1$.

Revenons à nos fonctions logiques et utilisons notre interprétation géométrique. Les fonctions qui sont modélisables par un perceptron simple sont celles dont les valeurs de 0 peuvent être séparées des valeurs de 1 en utilisant une droite. Par exemple, il est évident que les fonctions *et* et *ou* font partie des fonctions qui peuvent être modélisées. La figure 2.15 illustre la séparation de l'espace d'entrées pour ces deux fonctions.

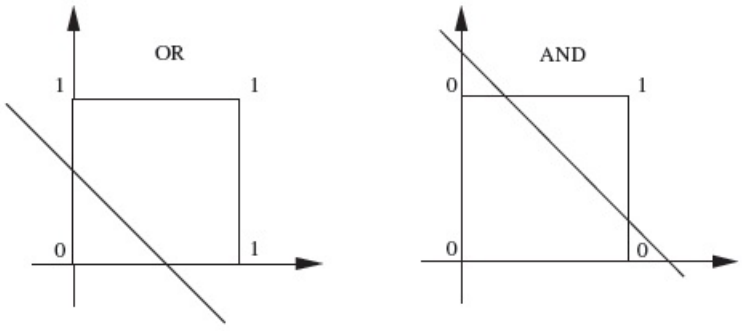

FIGURE 2.15 – Séparation des espaces d'entrées des fonctions logiques *ou* et *et*. Les points situés au-dessus des droites donnent une sortie de 1 tandis que ceux situés sous celle-ci ont une sortie égale à 0.

Les entrées des fonctions *xor* et *identité* vues précédemment ne peuvent pas être séparées de cette façon (voir figure 2.16). Voici donc la propriété que nous cherchions pour expliquer le fait qu'elles ne soient pas modélisables par un perceptron simple.

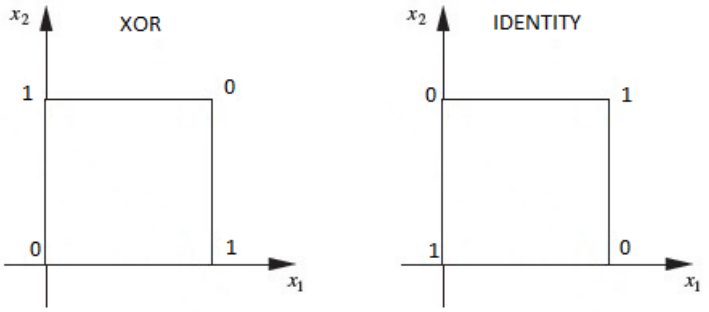

FIGURE 2.16 – Espaces d'entrées des fonctions logiques *xor* et *identité*. Dans ces deux espaces, les valeurs de 1 ne peuvent pas être séparées des valeurs de 0 à l'aide d'une droite.

Avec nos fonctions logiques, nous avons introduit la notion de séparabilité linéaire.

Définition 2.3.2 *Deux ensembles de points A et B dans un espace à n dimensions sont linéairement séparables s'il existe $n+1$ nombres réels $w_1, \ldots, w_{n+1}$ tels que chaque point $(x_1, x_2, \ldots, x_n) \in A$ satisfait $\sum_{i=1}^{n} w_i x_i \geq w_{n+1}$ et que chaque point $(y_1, y_2, \ldots, y_n) \in B$ satisfait $\sum_{i=1}^{n} w_i y_i < w_{n+1}$.*

Définition 2.3.3 *Une fonction est linéairement séparable si ses vecteurs d'entrées peuvent être groupés en deux ensembles linéairement séparables.*

Nous avons alors une proposition qui énonce le résultat que nous venons d'obtenir.

Proposition 2.3.1 *Un perceptron simple ne peut modéliser que les fonctions linéairement séparables.*

Nous pouvons à présent nous demander quelle est la proportion de fonctions linéairement séparables parmi les fonctions logiques à n arguments. La solution n'est pas connue. En effet, « Pour $n = 2$, nous avons 14 fonctions sur 16 qui sont linéairement séparables. Quand $n = 3$, nous passons à 104 sur 256 et quand $n = 4$, 1882 sur 65536 fonctions existantes sont linéairement séparables. Actuellement, aucune formule générale n'est connue pour exprimer le nombre de fonctions à n arguments linéairement séparables » (Rojas [24]).

2.3.4 Réseaux de neurones : le perceptron

Nous allons maintenant développer les réseaux de neurones qui peuvent être construits en associant des perceptrons simples (voir figure 2.17). Dans ce modèle, les réseaux sont formés de trois couches. La première couche, appelée rétine,

est une couche d'entrées qui fournit des informations à la deuxième. Celle-ci, appelée couche cachée, calcule les informations obtenues et renvoie un résultat à la troisième couche qui est la couche de sortie.

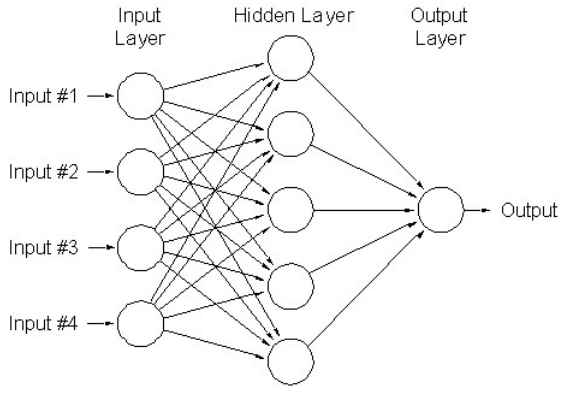

FIGURE 2.17 – Représentation du perceptron. Les réseaux de neurones de ce type sont formés de trois couches : la rétine (couche d'entrées), la couche cachée et la couche de sorties.

Ce modèle est le modèle de base du perceptron. A partir de celui-ci, nous pouvons développer un grand nombre de réseaux. Par exemple, nous pouvons ajouter une couche de neurones ou nous pouvons choisir de ne pas connecter toutes les entrées à chaque neurone de la couche cachée.

Comme pour le modèle de McCulloch-Pitts, les réseaux de neurones construits sur le modèle du perceptron sont capables de modéliser toutes les fonctions booléennes. De plus, les réseaux utilisés lors de ces modélisations seront plus simples que ceux du modèle précédent puisqu'ils sont plus malléables. En effet, en plus des seuils, nous pouvons changer les poids du réseau pour maximiser ses performances. Un autre avantage de ce modèle est que nous pouvons lui appliquer un processus d'apprentissage. Il existe plusieurs algorithmes d'apprentissage et nous parlerons plus longuement de ceux-ci par la suite.

2.4 Classification des réseaux de neurones

Au cours du développement du modèle général ainsi que des modéles de McCulloch-Pitts et du perceptron, nous avons pu remarquer certaines différences entre les réseaux de neurones. Nous allons à présent donner une classification plus systématique de nos réseaux en fonction de leurs caractéristiques.

Nous allons avoir deux types de classification. Tout d'abord, nous allons observer des réseaux de neurones différents mais qui sont équivalents dans le sens où ils sont capables de modéliser les mêmes fonctions. Nous passerons ensuite à une classification de réseaux qui ne seront pas équivalents.

2.4.1 Réseaux équivalents

Comme nous l'avons annoncé précédemment, les réseaux de neurones équivalents sont des réseaux qui ont des propriétés différentes mais qui sont capables d'exécuter les mêmes tâches, c'est-à-dire de modéliser les mêmes fonctions. Nous allons détailler ici deux caractéristiques des réseaux. La première est la présence ou l'absence de poids sur les connexions des neurones. La seconde est une différence dans la façon de considérer les arêtes inhibitrices.

Réseaux avec poids et sans poids

Les réseaux de neurones de McCulloch-Pitts n'ont pas de poids sur leurs connexions tandis que les perceptrons en ont. Nous allons montrer que si les poids sont des nombres rationnels, il y a équivalence entre ces deux types de réseaux. Nous allons illustrer cette équivalence à l'aide d'un exemple.

Dans la figure 2.18, le neurone avec poids modélise l'inégalité

$$0.2x_1 + 0.4x_2 + 0.3x_3 \geq 0.7$$

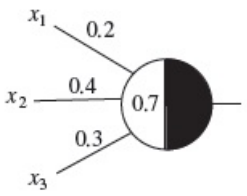

FIGURE 2.18 – Exemple de neurone avec poids. Ce neurone modélise l'inégalité $0.2x_1 + 0.4x_2 + 0.3x_3 \geq 0.7$.

Mais il est évident que la modéliser équivaut à modéliser

$$2x_1 + 4x_2 + 3x_3 \geq 7$$

Cette seconde inégalité peut être modélisée par un neurone sans poids en multipliant les arêtes du réseau le nombre de fois nécessaires (voir figure 2.19).

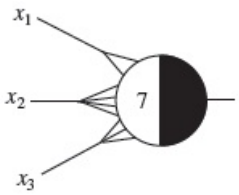

FIGURE 2.19 – Exemple de neurone sans poids. Ce neurone modélise l'inégalité $2x_1 + 4x_2 + 3x_3 \geq 7$ et est équivalent au neurone avec poids de la figure 2.18.

Nous pourrons donc modéliser les mêmes fonctions à l'aide de neurones avec ou sans poids.

Proposition 2.4.1 *Si les poids sont des rationnels, il est équivalent d'utiliser un réseau de neurones avec ou sans poids pour modéliser une fonction.*

Remarque
Il est important de noter que le fait d'utiliser un réseau de neurones sans poids va compliquer la structure de celui-ci. En effet, il aura plus d'arêtes et cela peut vite devenir gênant dans sa représentation. Pour des raisons de simplicité, nous préfèrerons donc utiliser des réseaux avec poids pour modéliser des fonctions plus compliquées.

Inhibition absolue et relative

Dans le cas des neurones de McCulloch-Pitts, nous avons travaillé avec une règle d'inhibition absolue. En effet, si une arête inhibitrice envoyait un signal de valeur 1, la sortie du neurone était automatiquement 0.

Il est également possible de travailler avec une règle d'inhibition relative. Celle-ci correspond à l'insertion d'un poids négatif sur l'arête, ce qui la rend inhibitrice et entraîne une diminution de la somme qui est comparée au seuil d'excitation du neurone. Dans ce cas, il est possible pour le neurone d'avoir une sortie de 1 même s'il possède des arêtes inhibitrices de valeur 1. Cette inhibition est celle utilisée par les perceptrons simples.

Ces deux types d'inhibition vont donner des réseaux de neurones équivalents.

Proposition 2.4.2 *Les réseaux utilisant une règle d'inhibition absolue et ceux utilisant une règle d'inhibition relative sont capables de modéliser les mêmes fonctions.*

Pour illustrer cette propriété, nous allons montrer que les réseaux de neurones de la figure 2.20 sont équivalents. Le réseau de gauche est un réseau avec inhibition relative tandis que celui de droite est un réseau avec inhibition absolue.

Ces deux réseaux ont des architectures totalement différentes. Le réseau de gauche est composé d'une seule couche tandis que celui de droite en a deux. Dans la figure de droite, nous pouvons voir que l'entrée y n'est pas utilisée par le second neurone de la première couche.

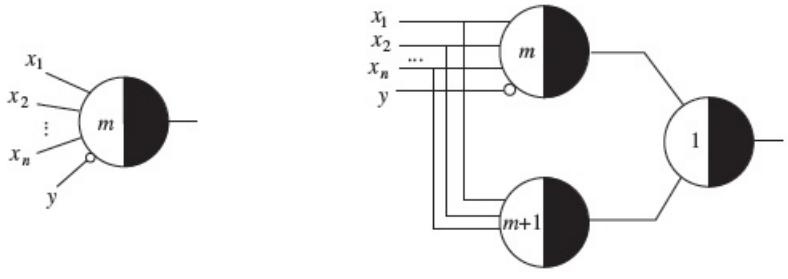

FIGURE 2.20 – Comparaison entre inhibition absolue et relative. Ces deux réseaux modélisent la même fonction. La règle d'inhibition absolue est utilisée dans le réseau de droite tandis que celui de gauche est soumis à la règle d'inhibition relative.

Si nous calculons la table de sorties du réseau de gauche pour $n = 2$ et $m = 1$, nous obtenons les résultats de la table 2.8. Nous obtenons la même table pour le réseau de droite si nous prenons les mêmes valeurs pour n et m. Cela signifie que nos deux réseaux modélisent bien la même fonction.

x_1	x_2	y	Sortie
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	0
1	1	1	1

TABLE 2.8 – Table de sorties des réseaux de neurones représentés dans la figure 2.20. Comme ces réseaux ont la même table de sorties, ils modélisent la même fonction.

2.4.2 Réseaux non équivalents

Les réseaux de neurones non équivalents sont des réseaux qui ne peuvent pas effectuer les mêmes tâches. Cependant, les différentes caractéristiques données ci-dessous sont très importantes dans le fonctionnement du neurone.

Réseaux feedforward et récurrents

Tout au long de ce chapitre, nous avons travaillé sur des réseaux de neurones feedforward, c'est-à-dire sur des réseaux de neurones qui ne contiennent pas de cycles. Ils ont un sens déterminé et l'information ne passe pas deux fois par le même neurone. Dans le cas où les réseaux de neurones contiennent des cycles, nous parlerons de réseaux récurrents (voir figure 2.21).

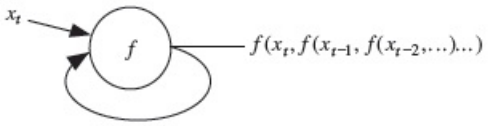

FIGURE 2.21 – Exemple de réseau de neurones récurrent. Ce type de réseau contient des cycles, ce qui signifie que le signal passe plusieurs fois par le même neurone.

Pour ce type de réseaux, nous devons tenir compte de plus de paramètres que dans les réseaux que nous avons étudiés précédemment. En effet, nous allons devoir nous poser les deux questions suivantes :

- Comment peut-on savoir que nous avons le résultat final et pas un résultat intermédiaire ?
- Comment peut-on accéder à l'information contenue dans un niveau intermédiaire ?

Pour savoir si nous avons bien obtenu le résultat final, nous allons devoir introduire une notion de temps. Si un cycle est effectué au temps t, son résultat est connu au temps $t+1$ et nous allons fixer un temps final auquel nous connaîtrons la solution finale.

En ce qui concerne l'information contenue dans les niveaux intermédiaires, nous allons devoir introduire des outils de stockage pour pouvoir y accéder.

Nous pouvons voir par les réponses aux deux questions précédentes qu'il sera plus difficile de modéliser un réseau de neurones récurrent car il faudra faire intervenir d'autres éléments que ceux présentés pour les réseaux feedforward.

Réseaux synchrones et asynchrones

Les réseaux synchrones sont des réseaux dans lesquels tous les neurones d'une même couche donnent leur solution de manière simultanée. Jusqu'à présent, nous avons toujours travaillé avec ce type de réseaux.

Il est cependant possible de travailler avec des réseaux asynchrones. Comme son nom l'indique, ce type de réseaux n'a pas des neurones qui donnent leur réponse en même temps. Au contraire, chacun d'eux donne sa sortie indépendamment des autres.

2.5 Conclusion

Nous venons d'étudier la modélisation de nos réseaux de neurones. Pour cela, nous avons introduit une modélisation générale de nos réseaux ainsi que deux exemples de modèles possédant des caractéristiques différentes. Pour chacun d'eux, nous avons illustré leur utilisation et leur interprétation géométrique sur les fonctions booléennes. A partir des différences observées dans la conception de nos deux modèles, nous avons effectué une classification des modèles de réseaux de neurones. Cette classification n'a pas pour but d'être exhaustive. Par celle-ci, nous voulons simplement montrer aux lecteurs l'abondance de modèles différents pouvant être construits.

Nous aurions donc pu développer un grand nombre d'autres modèles de réseaux de neurones, dont certains beaucoup plus complexes que les deux présentés. Nous décidons toutefois de ne pas nous attarder plus longuement sur la modélisation de nos réseaux et de passer à l'apprentissage de ceux-ci.

Chapitre 3

Apprentissage classique des réseaux de neurones

A présent que nous avons détaillé la modélisation de nos réseaux de neurones, nous allons pouvoir introduire le concept d'apprentissage lié à ceux-ci. Nous commencerons par expliciter ce que nous voulons dire par apprentissage de nos réseaux. Nous donnerons ensuite deux exemples d'algorithmes utilisés pour réaliser ce type d'apprentissage. Nous tirerons finalement une petite conclusion sur ce que nous avons appris à l'aide de ces algorithmes et sur ce que nous souhaitons faire dans la suite de ce travail.

Les principales références pour ce chapitre sont celles de MacKay [19], Peretto [21] et Rojas [24].

3.1 Généralités

Dans le chapitre précédent, nous avons travaillé sur deux modèles de réseaux de neurones. Toutefois, nous n'avons pas abordé la façon de trouver les poids et les seuils qui permettent de modéliser une fonction particulière. En effet, jusqu'à présent, nous avons donné ces valeurs en les déterminant au cas par cas en fonction de la modélisation à effectuer. Nous aimerions à présent trouver une méthode pour déterminer ces seuils et poids de façon automatique. Pour ce faire, nous utiliserons des algorithmes particuliers appelés algorithmes d'apprentissage. Leur fonctionnement est illustré par la figure 3.1.

Les algorithmes d'apprentissage sont des méthodes adaptatives. Ils ont pour but de faire évoluer les poids (et/ou les seuils) du réseau afin que celui-ci modélise la fonction désirée. Il existe différents types d'apprentissage. En effet, les algorithmes d'apprentissage peuvent être divisés en deux grandes classes : les méthodes supervisées et non supervisées.

L'apprentissage supervisé consiste à présenter différents vecteurs d'entrées au réseau. La sortie calculée par le réseau est ensuite comparée à celle attendue, appelée la cible. En cas d'erreur, les poids sont modifiés. Ce type d'apprentissage peut-être subdivisé suivant le niveau de connaissance de l'erreur. Ces deux sub-

divisions sont connues comme apprentissage par renforcement et par correction d'erreur.

- L'apprentissage par renforcement est requis lorsque le renseignement obtenu pour l'erreur est booléen, c'est-à-dire lorsque l'utilisateur sait uniquement si la sortie du réseau est celle attendue ou non. Dans ce cas, l'utilisateur ne connaît pas l'amplitude de l'erreur commise et ne peut utiliser que les entrées pour corriger les poids.
- L'apprentissage par correction est utilisé lorsque l'utilisateur connaît l'amplitude de l'erreur. Cette connaissance supplémentaire lui permet de déterminer l'ampleur de la correction à apporter. En général, l'erreur peut alors être corrigée en une seule étape.

L'apprentissage supervisé est aussi appelé apprentissage avec un professeur car les sorties associées aux différentes configurations d'entrées sont connues.

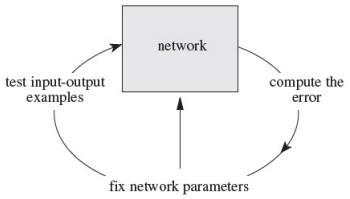

FIGURE 3.1 – Fonctionnement des algorithmes d'apprentissage. Une série d'exemples d'entrées associées à leur sortie attendue sont envoyés dans le réseau. L'erreur entre la sortie et la sortie attendue est calculée et les paramètres du réseau sont modifiés afin de miniser cette erreur. Source : [24].

L'apprentissage non supervisé, quant à lui, est utilisé quand la réponse exacte qu'un réseau devrait produire pour un vecteur d'entrées donné est inconnue. C'est le cas par exemple si l'utilisateur souhaite classer des points. En général, le nombre de classes lui-même n'est pas connu.

3.2 Exemples d'algorithmes d'apprentissage

Nous allons présenter deux exemples d'algorithmes d'apprentissage. Le premier est l'algorithme d'apprentissage du perceptron et fait partie des méthodes supervisées. Le second est l'algorithme d'apprentissage par compétition et appartient à la classe des méthodes non supervisées.

3.2.1 Algorithme d'apprentissage du perceptron

Comme annoncé plus tôt, l'algorithme d'apprentissage du perceptron est un exemple d'apprentissage supervisé. La version que nous allons présenter fait partie des méthodes d'apprentissage par renforcement. Il est toutefois possible de trouver des versions de cet algorithme qui utilisent un apprentissage par correction d'erreurs. Nous allons commencer par introduire certains concepts nécessaires à sa compréhension et écrire l'algorithme en lui-même. Nous donnerons ensuite son interprétation géométrique et aborderons finalement le sujet de sa convergence.

Préliminaires

Commençons par introduire une notation qui nous permettra de simplifier l'écriture de notre algorithme. Nous définissons le vecteur d'entrées $x = (x_1, x_2, \ldots, x_n)$ et le vecteur de poids $w = (w_1, w_2, \ldots, w_n)$. De cette façon, notre excitation totale pourra s'écrire sous forme de produit scalaire $w \cdot x$ plutôt que sous la forme d'une somme.

Nous avons vu que le perceptron simple permet de représenter les fonctions qui sont linéairement séparables. Nous allons à présent définir les fonctions absolument linéairement séparables.

Définition 3.2.1 *Deux ensembles de points A et B dans un espace à n dimensions sont absolument linéairement séparables s'il existe $n + 1$ nombres réels $w_1, w_2, \ldots, w_n$ et θ tels que tous points $x = (x_1, x_2, \ldots, x_n) \in A$ vérifient $w \cdot x > \theta$ et tous points $x = (x_1, x_2, \ldots, x_n) \in B$ vérifient $w \cdot x < \theta$.*

Définition 3.2.2 *Une fonction est absolument linéairement séparable si ses vecteurs d'entrées peuvent être séparés en deux ensembles absolument linéairement séparables.*

Un perceptron simple capable de séparer linérairement deux ensembles finis de vecteurs d'entrées est aussi capable de séparer absolument linéairement ces ensembles. En effet, il suffit d'ajuster légèrement les poids de la séparation linéaire pour obtenir une séparation linéaire absolue. C'est un corollaire direct de la proposition suivante.

Proposition 3.2.1 *Deux ensembles finis de points A et B dans un espace à n dimensions qui sont linéairement séparables sont aussi absolument linéairement séparables.*

Comme nous avons vu que le perceptron simple est capable de modéliser toutes les fonctions linéairement séparables et vu l'équivalence entre les ensembles linéairement séparables et absolument linéairement séparables, le perceptron simple va donc être capable de modéliser toutes les fonctions absolument linéairement séparables. Cette petite modification dans la façon de définir les fonctions modélisables par le perceptron simple nous permettra de prouver la convergence de notre algorithme sous certaines conditions.

Nous allons maintenant introduire une nouvelle notation. Comme nous l'avons déjà annoncé, le perceptron sépare les vecteurs d'entrées en deux ensembles. Pour rappel, trier les vecteurs d'entrées de façon correcte équivaut à retrouver la table de vérité de la fonction à modéliser. Nous allons noter P, l'ensemble des vecteurs qui doivent vérifier $w \cdot x > \theta$ et N, l'ensemble des vecteurs qui doivent vérifier la relation contraire, c'est-à-dire $w \cdot x < \theta$. L'ensemble P correspond donc à l'ensemble des vecteurs d'entrées auxquels correspond une sortie égale à 1 et N correspond à l'ensemble des vecteurs d'entrées auxquels correspond une sortie de 0.

Nous disposons à présent de toutes les notations et de tous les concepts nécessaires pour comprendre le fonctionnement de l'algorithme d'apprentissage du perceptron simple.

Algorithme

Pour rappel, le but de cet algorithme est de séparer correctement les vecteurs d'entrées. Cela est possible uniquement si le perceptron simple modélise une fonction linéairement séparable. Nous allons écrire un pseudo-code de l'algorithme.

Algorithme 3.2.1 *Apprentissage du perceptron simple*

- *Initialisation : le vecteur de poids w et le seuil θ sont générés de façon aléatoire*
- *Itération : un vecteur $x \in P \cup N$ est sélectionné de façon aléatoire*
  - ⋆ *si $x \in P$ et $w \cdot x \leq \theta$, alors*
    $w = w + x$
    $\theta = \theta - 1$
  - ⋆ *si $x \in N$ et $w \cdot x \geq \theta$, alors*
    $w = w - x$
    $\theta = \theta + 1$
  - ⋆ *sinon, le vecteur x est replacé dans $P \cup N$ et on choisit de nouveau un vecteur de façon aléatoire dans cet ensemble.*
- *Terminaison : tous les vecteurs sont bien classés*

Comme indiqué dans le pseudo-code, la première étape consiste à choisir un vecteur de poids et un seuil de façon aléatoire. Dans la boucle, une correction du vecteur de poids est effectuée si le vecteur d'entrées sélectionné est mal classé. L'algorithme se termine lorsque tous les vecteurs sont bien classés.

Remarque
Il est parfois nécessaire qu'un vecteur soit sélectionné plusieurs fois avant d'être bien classé. En effet, cet apprentissage est un apprentissage par renforcement et l'importance de l'erreur commise n'est pas connue. Une seule mise à jour d'un vecteur peut donc ne pas être suffisante pour le classer correctement.

Interprétation géométrique

Nous allons donner une interprétation géométrique de l'objectif de notre algorithme. Ensuite, nous illustrerons la façon dont il réalise sa tâche. Dans cette interprétation, nous allons supposer que nous travaillons avec un seuil nul. Ce cas est très facile à obtenir en modifiant légèrement notre réseau comme nous l'avons vu dans le chapitre précédent (représentation dans la figure 2.13). Cela nous permettra de simplifier notre interprétation.

L'algorithme a pour objectif d'optimiser le vecteur de poids afin que les points des ensembles P et N vérifient la bonne relation. Le processus utilisé dans ce but est représenté dans la figure 3.2.

Dans cet algorithme, l'erreur du perceptron avec un vecteur de poids w correspond au nombre de points mal classés. L'algorithme d'apprentissage du perceptron simple essaie donc de minimiser cette fonction d'erreur. Dans ce but, il cherche une direction vers laquelle se déplacer dans l'espace des poids pour diminuer cette erreur. Il fait ensuite une mise à jour du vecteur de poids dans cette

direction. La direction choisie est celle du vecteur d'entrées qui est sélectionné à chaque itération de la boucle (s'il est mal classé).

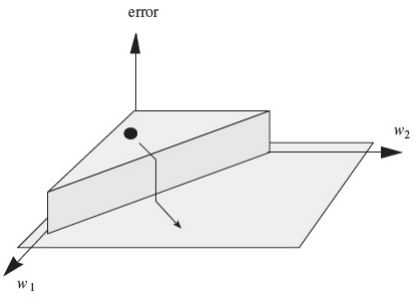

FIGURE 3.2 – Exemple de surface d'erreur et direction de descente. Le vecteur de poids est modifié de façon à minimiser l'erreur commise lors de la modélisation. La mise à jour du vecteur est faite dans la direction de descente. Source : [24].

Si un vecteur $x \in P$ tel que $w \cdot x < 0$ est trouvé, cela signifie que l'angle entre les deux vecteurs est plus grand que 90 degrés. Le vecteur de poids doit donc être modifié dans la direction du vecteur x pour diminuer l'amplitude de cet angle. Cette modification est effectuée par l'addition de x et w. Au contraire, si $x \in N$ et $w \cdot x > 0$, alors l'angle entre x et w est inférieur à 90 degrés. Le vecteur de poids doit donc être pivoté dans la direction opposée à x, ce qui est fait en soustrayant x à w. La figure 3.3 illustre ce processus dans le cas où x_1 doit appartenir à P.

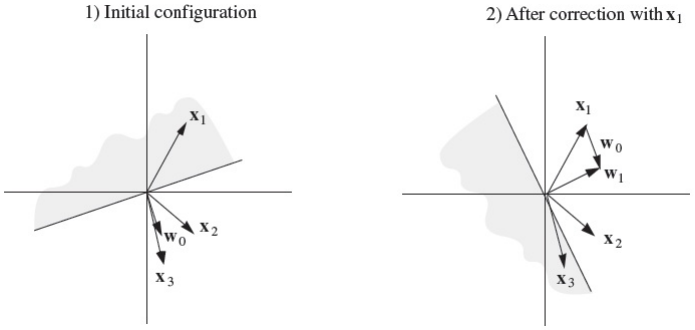

FIGURE 3.3 – Fonctionnement de l'algorithme d'apprentissage du perceptron. L'illustration de gauche représente la configuration initiale. Celle de droite représente la configuration après correction du vecteur de poids en fonction de x_1. Le vecteur x_1 doit appartenir à P. Sinon, il n'y aurait pas eu de correction car l'angle était supérieur à 90 degrés. Source : [24].

Dans l'algorithme d'apprentissage du perceptron simple que nous venons de voir, nous ne travaillons pas avec des vecteurs normalisés. Cela implique que la longueur du vecteur de poids augmente à chaque correction. Cette croissance est également représentée dans la figure 3.3. Il est toutefois envisageable de modifier notre algorithme de façon à travailler avec des vecteurs normalisés.

Convergence et algorithme modifié

Nous allons à présent aborder le problème de la convergence de l'algorithme d'apprentissage du perceptron. Si la fonction à modéliser possède des ensembles de vecteurs d'entrées absolument linéairement séparables, il est possible de montrer que l'algorithme va mettre à jour le vecteur de poids un nombre fini de fois. Cela signifie que l'algorithme converge vers un vecteur de poids optimal, capable de classer tous les vecteurs d'entrées dans le bon ensemble. C'est ce qui est exprimé dans la proposition de convergence qui suit.

Proposition 3.2.2 *Si les ensembles P et N sont finis et absolument linéairement séparables, l'algorithme d'apprentissage du perceptron met à jour le vecteur de poids un nombre fini de fois. En d'autres mots, si les vecteurs dans P et N sont testés de façon cyclique les uns après les autres, un vecteur de poids w qui sépare les deux ensembles est trouvé après un nombre fini d'étapes.*

Comme nous l'avons vu plus tôt, toutes les fonctions n'ont pas des vecteurs d'entrées pouvant être divisés en deux ensembles absolument linéairement séparables. Si nous essayons de modéliser une de ces fonctions et cherchons le vecteur de poids à l'aide de l'algorithme étudié précédemment, nous n'aurons jamais de solution. En effet, cet algorithme ne se terminera jamais puisqu'il ne pourra jamais séparer les vecteurs d'entrées en deux ensembles absolument linéairement séparables.

Lorsqu'aucune séparation parfaite n'existe, l'objectif est de calculer le vecteur de poids qui sépare correctement le plus grand nombre de vecteurs possibles dans P et N. Pour cela, une variante de l'algorithme d'apprentissage du perceptron simple a été proposée par Gallant. L'idée principale de cet algorithme est de garder en mémoire le meilleur vecteur de poids, c'est-à-dire le vecteur de poids qui sépare correctement le plus grand nombre de vecteurs d'entrées, tout en continuant à le mettre à jour. Si la mise à jour aboutit à un meilleur vecteur, celui-ci remplace l'ancien dans la mémoire et l'algorithme continue. Cet algorithme a pour nom l'algorithme de poche.

Algorithme 3.2.2 *Algorithme de poche*

- *Initialisation : le vecteur de poids w et le seuil θ sont générés aléatoirement $h_s = 0$, $w_s = w$ et $\theta_s = \theta$.*
- *Itération : la mise à jour de w se fait en utilisant une seule itération de l'algorithme d'apprentissage du perceptron simple. On garde une trace du nombre h de vecteurs testés de façon consécutive avec succès. Si, à n'importe quel moment $h > h_s$, on remplace w_s par w, θ_s par θ et h_s par h.*

Comme nous l'avons déjà dit, l'algorithme d'apprentissage du perceptron est, de façon évidente, un exemple d'apprentissage supervisé. D'un point de vue biologique, l'approche se cachant derrière cet apprentissage ne semble pas très logique. En effet, elle nécessite le présence d'un professeur qui sait au minimum si la sortie est celle cherchée ou non. Dans la réalité, nous ne connaissons pas toujours les sorties à l'avance. Par exemple, lorsque nous cherchons à classer des éléments, nous ne pouvons pas dire d'avance dans quelle classe chaque élément va se trouver.

Nous allons à présent développer une méthode d'apprentissage non supervisée dans laquelle les poids du réseau sont déterminés par un processus d'organisation interne. Nous pouvons distinguer deux classes d'apprentissage non supervisé appelées apprentissage par renforcement et apprentissage par compétition. Dans l'apprentissage par renforcement, chaque entrée modifie les poids afin d'améliorer le comportement interne du réseau. La règle hebbienne est une des règles qui peut être utilisée lors de ce type d'apprentissage. L'idée qui se cache derrière celle-ci est que deux neurones en activité au même moment vont créer ou renforcer leur connexion de façon à faciliter l'activation de l'un par l'autre lors de futures itérations. Dans l'apprentissage compétitif, les neurones entrent en compétition pour obtenir le droit de fournir la sortie associée à un vecteur d'entrées. Quand un des neurones gagne ce droit, il rend tous les autres neurones inhibiteurs. Nous allons maintenant développer ce principe d'apprentissage par compétition dans un cadre se rapportant à celui de l'algorithme d'apprentissage du perceptron.

3.2.2 Algorithme d'apprentissage par compétition

La méthode d'apprentissage non supervisée par compétition est basée, comme annoncé précédemment, sur la compétition entre les neurones du réseau. Pour comprendre son fonctionnement, nous allons tout d'abord introduire quelques concepts préliminaires ainsi que le cadre dans lequel nous allons utiliser cet apprentissage. Nous donnerons ensuite un pseudo-code de cet algorithme avant de parler brièvement de son interprétation géométrique et de sa convergence.

Préliminaires

Commençons par introduire le problème auquel nous voulons appliquer notre apprentissage. La situation sur laquelle nous travaillons est semblable à celle présentée lors de l'étude de l'algorithme d'apprentissage du perceptron simple. En effet, notre but est, à nouveau, de séparer des vecteurs d'entrées en classes. Toutefois, contrairement à la situation précédente, ces vecteurs ne se séparent plus en deux ensembles absolument linéairement séparables et ne peuvent donc plus être séparés par un perceptron simple. La figure 3.4 illustre à quoi peuvent ressembler de tels ensembles dans un espace à deux dimensions.

Pour pouvoir résoudre ce problème, nous allons devoir utiliser plusieurs neurones qui vont entrer en compétition pour donner la sortie. Nous pouvons voir sur le graphe de la figure 3.4 que les vecteurs d'entrées se séparent naturellement en trois classes. Nous aurons donc besoin de trois neurones pour modéliser notre problème. Le réseau formé dans ce but est représenté dans la figure 3.5. Nous pouvons voir que trois vecteurs de poids $w_1 = (w_{11}, w_{12}), w_2 = (w_{21}, w_{22})$ et $w_3 = (w_{31}, w_{32})$ ont été défini pour correspondre aux trois neurones du réseau. Nous allons à présent analyser ce qui se passe dans l'algorithme et en donner une interprétation géométrique.

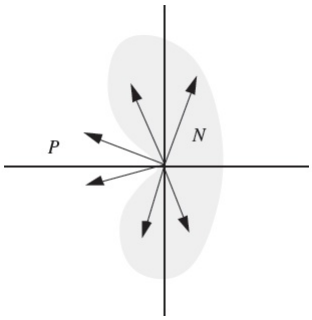

FIGURE 3.4 – Problème de classification. L'objectif est de séparer ces deux ensembles de vecteurs d'entrées. Nous pouvons facilement voir que ceux-ci ne sont pas linéairement séparables. L'ensemble N est composé de vecteurs partant dans deux directions différentes (réunion de deux classes). Source : [24].

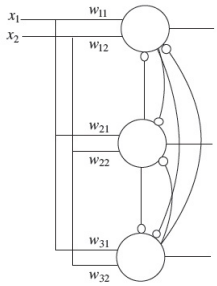

FIGURE 3.5 – Réseau de trois neurones en compétition. A chaque neurone est associé un vecteur de poids. Pour rappel, les arêtes représentées avec un rond vide sont des arêtes inhibitrices. Lors de l'exécution d'un vecteur d'entrées particulier, un seul neurone est excité et donne la sortie du réseau. Les deux autres, quant à eux, sont inhibés. Source : [24].

Algorithme

L'algorithme développé ci-dessous nous permet d'identifier les différentes classes existant au sein de l'ensemble des vecteurs d'entrées. Sans perte de généralité, nous pouvons considérer que chacun des neurones a un seuil nul. Ecrivons à présent un pseudo-code de cet algorithme.

Algorithme 3.2.3 *Algorithme d'apprentissage par compétition*
Soit $X = \{X_1, X_2, \ldots, X_l\}$, un ensemble de vecteurs d'entrées normalisés dans l'espace à n dimensions que nous voulons classer en k classes. Le réseau se compose de k neurones, chacun d'eux étant composé de n entrées et d'un seuil nul.

- *Initialisation : générer aléatoirement les vecteurs normalisés* $W_1, W_2, \ldots, W_k \in \mathbf{R}^n$

- *Itération :*
  - ⋆ *sélectionner un vecteur* $X_j \in X$ *de façon aléatoire*
  - ⋆ *calculer* $X_j \cdot W_i$ *pour* $i = 1, \ldots, k$
  - ⋆ *sélectionner* W_m *tq* $W_m \cdot X_j \geq W_i \cdot X_j$ *pour* $i = 1, \ldots, k$
  - ⋆ *mettre à jour* $W_m = W_m + X_j$ *et normaliser*

Comme dans l'algorithme d'apprentissage du perceptron simple, la première étape consiste à choisir les vecteurs de poids de façon aléatoire. Dans la boucle, une correction est effectuée pour le vecteur de poids correspondant au neurone pour lequel l'excitation est la plus grande. Nous pouvons remarquer que cet algorithme ne possède pas de condition de terminaison. Toutefois, « L'algorithme peut être arrêté après un nombre prédéterminé d'étapes » (Rojas [24]).

Dans ce pseudo-code, la mise à jour du vecteur de poids se fait en ajoutant le vecteur d'entrée à l'ancien vecteur de poids. Cependant, il existe d'autres façons de le modifier. Voici une description des méthodes les plus utilisées.

- Mise à jour avec constante d'apprentissage.
  La mise à jour du poids est donnée par

  $$w_m = w_m + \eta x_j$$

  où le paramètre d'apprentissage η est un nombre réel entre 0 et 1. Ce paramètre décroît au fur et à mesure de la progression de l'algorithme. Les corrections effectuées sont donc plus importantes au début du processus qu'à sa fin.
- Mise à jour par différence.
  Le poids est mis à jour en suivant l'équation suivante

  $$w_m = w_m + \eta(x_j - w_m)$$

  où le paramètre η est de nouveau un nombre réel entre 0 et 1. Dans ce cas, la correction effectuée sur le vecteur de poids est proportionnelle à la différence entre celui-ci et le vecteur d'entrées sélectionné.
- Mise à jour par lots.
  Plutôt que d'être ajoutées à chaque itération, les mises à jour à appliquer aux vecteurs de poids sont calculées et conservées. Elles sont toutes ajoutées en même temps après un nombre donné d'itérations. Cette méthode de mise à jour garantit une certaine stabilité dans le processus d'apprentissage.

Interprétation géométrique

Nous allons maintenant tenter d'expliquer notre algorithme d'un point de vue géométrique. La situation est représentée par la figure 3.6.

Nous avons vu dans la figure 3.4 que les vecteurs des deux ensembles étaient séparés en trois classes. Le but est que les trois neurones du réseau soient capables de placer un maximum de vecteurs d'entrées dans la bonne classe. Dans ce cas, nous pouvons considérer le vecteur de poids associé à un des 3 neurones

comme un représentant des vecteurs contenus dans la classe correspondant à ce neurone.

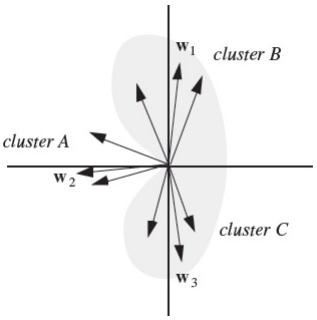

FIGURE 3.6 – Relation entre classes et vecteurs de poids. Le but de notre réseau est de classer correctement un maximum de vecteurs d'entrées. Pour cela, le vecteur de poids de chaque neurone est considéré comme un représentant d'une classe en particulier. Comme l'angle entre un vecteur appartenant à une classe et son vecteur de poids est plus petit que 90 degrés, le produit scalaire entre les deux sera positif. De plus, plus l'angle est petit, plus le produit scalaire est grand. Nous devons donc bien prendre le vecteur de poids avec lequel le produit scalaire est le plus grand dans l'algorithme. Source : [24] .

Lorsqu'un vecteur appartient sans perte de généralité à la classe A, son produit scalaire avec le vecteur de poids w_2 est positif. En effet, l'angle entre les deux vecteurs est inférieur à 90 degrés. De plus, nous travaillons avec des vecteurs normalisés et, dans ce cas, le produit scalaire entre deux vecteurs correspond au cosinus de l'angle formé par ceux-ci. Donc, si nous avons deux vecteurs de poids qui nous donnent une valeur positive, nous savons qu'en prenant la plus grande valeur, nous prendrons le vecteur de poids le plus proche du vecteur d'entrées. C'est ce qui est fait dans l'algorithme développé ci-dessus. Le vecteur d'entrées est ensuite ajouté au vecteur de poids w_2 de façon à le faire tourner vers lui comme dans l'algorithme d'apprentissage du perceptron. Voyons à présent si cet algorithme converge.

Convergence

Contrairement à l'algorithme d'apprentissage du perceptron simple qui, pour rappel, converge toujours lorsque les deux ensembles de vecteurs d'entrées sont linéairement séparables, l'algorithme d'apprentissage par compétition ne converge pas toujours. En effet, il n'est pas évident de connaître à l'avance le nombre de classes contenues dans un ensemble de vecteurs. De ce fait, si nous utilisons un nombre de neurones inférieur au nombre de classes, l'algorithme va faire osciller nos vecteurs de poids de classes en classes dans une vaine tentative de séparation de l'espace. Au contraire, si nous utilisons plus de neurones que le nombre de classes présentes, deux cas de figure peuvent apparaître. Soit les vecteurs de poids de certains neurones ne sont jamais mis à jour et on parle alors de neurones morts. Soit nous nous retrouvons avec des sous-classes des classes naturelles.

La convergence dépend également de l'initialisation de nos vecteurs de poids. En effet, certaines configurations de départ nous permettent de retrouver nos classes tandis que d'autres configurations ne le permettent pas. Il est également à noter que l'algorithme effectue un nombre fixé d'itérations. Donc, pour de grands ensembles, nous parlons de convergence quand nous retrouvons à peu près nos classes naturelles mais il est évident que certaines erreurs persistent.

3.3 Conclusion

Ce chapitre nous a permis d'introduire la notion d'apprentissage classique ainsi que ses deux grandes subdivisions (méthodes supervisées et non supervisées). Nous avons pu voir que la convergence de ces méthodes n'est pas toujours assurée. Le but de ce travail n'est pas d'utiliser ces méthodes pour modéliser certaines fonctions mais plutôt d'utiliser une méthode alternative : les algorithmes génétiques. Le chapitre suivant sera donc consacré à ceux-ci et à la façon de les utiliser pour faire apprendre nos réseaux.

Chapitre 4

Algorithmes génétiques et apprentissage des réseaux

Le but de ce chapitre est d'analyser l'utilisation des algorithmes génétiques dans l'apprentissage de nos réseaux de neurones. Pour ce faire, nous allons tout d'abord introduire le concept d'algorithmes génétiques simples et multi-objectifs et en expliquer le fonctionnement. Nous verrons ensuite comment nous pouvons les utiliser pour améliorer les capacités de nos réseaux. Nous terminerons par une description de la façon dont nous avons implémenté nos réseaux et nos algorithmes génétiques.

Ce chapitre est principalement basé sur les références de Bodenhofer [4], de Deb [9] et [10] et de Goldberg [15].

4.1 Algorithmes génétiques simples

Les algorithmes génétiques sont des méthodes particulières d'optimisation. Comme nos réseaux de neurones artificiels, ils s'inspirent du fonctionnement d'un phénomène biologique naturel qui est l'évolution des espèces vivantes. Ce processus d'évolution repose sur la reproduction des individus, sur leur mutation et sur la sélection naturelle de leurs descendants. Ces trois principes vont permettre d'une part la diversification des espèces, c'est-à-dire l'exploration de l'espace des solutions, et d'autre part la sélection des individus les mieux adaptés à l'environnement, à savoir les solutions qui optimisent le problème.

Les algorithmes génétiques ont l'avantage d'être robustes et rapides. De plus, ils ne nécessitent aucune connaissance sur le système à optimiser ce qui représente un avantage certain par rapport aux méthodes classiques d'optimisation pour lesquelles la connaissance du gradient, par exemple, est indispensable. Il est toutefois possible d'améliorer les résultats des algorithmes génétiques si certains paramètres sont connus, ce qui nous permet de donner des méthodes adaptées au problème pour les différentes étapes de l'algorithme. Les algorithmes génétiques ont un coût plus important que ces méthodes classiques. De plus, ils ne peuvent donner qu'une solution approximative du problème et leur convergence est seulement assurée en probabilité.

Nous allons à présent définir la terminologie utilisée par ces algorithmes. Nous expliquerons également leur fonctionnement général et les principales étapes qui les composent.

4.1.1 Terminologie

Comme nous l'avons vu précédemment, les algorithmes génétiques s'inspirent d'un processus biologique naturel. Il est donc logique d'utiliser des termes issus de la biologie pour représenter les différents éléments de nos algorithmes. Nous allons donner une brève explication de chacun de ceux-ci.

- Un gène est une séquence d'ADN. Ce sont les informations génétiques contenues dans nos gènes qui vont déterminer nos caractéristiques anatomiques, moléculaires,... Par exemple, si nous simplifions, un gène peut contenir l'information génétique qui donnera la couleur des yeux d'un individu.
- Un allèle est une version possible pour un gène. Par exemple, les allèles possibles pour le gène correspondant à la couleur des yeux sont : bleu, vert, brun, noir,...
- Un chromosome est composé d'un certain nombre de gènes dont les valeurs sont les allèles.
- Un individu est constitué d'un ou plusieurs chromosomes.
- La population est définie comme un ensemble d'individus.
- La fitness est une mesure quantitative du niveau d'adaptation d'un individu à son environnement. C'est la mesure qui va permettre de réaliser la sélection naturelle.

Après avoir expliqué les principaux termes intervenant dans ces algorithmes génétiques, nous allons pouvoir en donner les étapes importantes.

4.1.2 Fonctionnement et schéma général

Le but des algorithmes génétiques est de trouver la solution optimale d'une fonction objectif f, c'est-à-dire les valeurs des variables qui vont maximiser (ou minimiser) cette fonction. Pour ce faire, ils font correspondre l'ensemble des solutions admissibles ([27]) du problème à un ensemble d'individus dont l'adaptabilité par rapport à leur environnement est calculée par une fonction de fitness qui correspond à la fonction f.

Les algorithmes génétiques génèrent une population aléatoire d'individus et simulent ensuite un processus d'évolution en alternant les quatre opérations détaillées ci-dessous.

- La sélection consiste à choisir les meilleurs individus pour la reproduction à partir de leur fitness.
- La reproduction permet d'obtenir de nouveaux individus (enfants) à partir des individus (parents) sélectionnés à l'étape précédente. Les enfants sont obtenus en mélangeant le code génétique des parents. Le but est d'obtenir des enfants au moins aussi bons que leurs parents.

- La mutation modifie légèrement le code génétique de certains individus. Cette opération a pour but d'apporter des infomations génétiques nouvelles dans la population, avec la possibilité que ces nouvelles informations améliorent le niveau d'adaptation par rapport à l'environnement. En terme d'optimisation, la mutation représente une tentative pour éviter une stagnation dans un optimum local.
- Le passage à la génération suivante est permis par la reproduction et la mutation qui engendre une nouvelle génération à partir de l'ancienne.

Ces opérations sont répétées jusqu'à une condition d'arrêt spécifiée. Celle-ci peut être un nombre atteint de générations ou un seuil d'adaptation suffisant. Le processus d'évolution décrit ci-dessus est représenté par la figure 4.1.

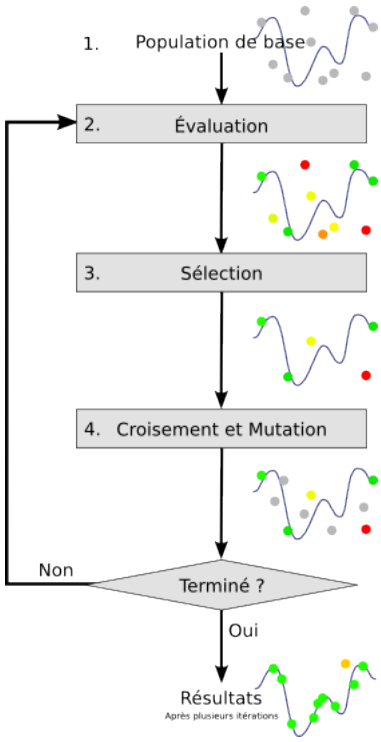

FIGURE 4.1 – Fonctionnement général des algorithmes génétiques. Ce schéma reprend les principales étapes de ces algorithmes, à savoir l'initialisation et les étapes à répéter : sélection, reproduction, mutation et passage à la génération suivante. Source : `http://fr.wikipedia.org/wiki/Algorithme_génétique`.

Le but est d'obtenir, au fil des générations et en suivant ces différentes étapes, des individus de mieux en mieux adaptés à leur environnement, ce qui correspond à des solutions admissibles de la fonction objectif f se rapprochant de plus en plus de la solution optimale. Nos générations successives nous permettent donc d'obtenir une solution approchée de l'optimum de la fonction f.

Nous venons de donner le mécanisme général des algorithmes génétiques. Cependant, il existe beaucoup de versions différentes de ces algorithmes suivant les méthodes de sélection, reproduction et mutation utilisées. Nous allons à présent nous attarder sur les principales méthodes qui existent pour chacune des opérations précitées.

4.1.3 Sélection

Comme nous l'avons énoncé plus tôt, la sélection est l'opération qui permet de choisir les meilleurs individus de la population pour la reproduction, ce qui entraîne une adaptabilité de plus en plus grande au fil des générations. Il existe plusieurs méthodes pour réaliser cette sélection. Nous allons décrire ici les plus connues d'entre-elles, à savoir la sélection par roulette, la "stochastic remainder without replacement" selection, le tournoi et l'élitisme.

Roulette ([15] pp.11-12)

La sélection par roulette, roulette wheel en anglais, consiste à associer à chaque individu un sous-intervalle du segment $[0, 1]$ de longueur proportionnelle à sa fitness. La sélection d'un individu peut ensuite se faire en tirant aléatoirement et de façon uniforme un nombre entre 0 et 1 et en choisissant l'individu correspondant au sous-intervalle qui contient ce nombre. Les individus les mieux adaptés auront plus de chance d'être choisis puisqu'ils correspondent à un plus grand sous-intervalle du segment $[0, 1]$.

Cette méthode est appelée sélection par roulette car chaque case de celle-ci pourrait représenter un individu et son étendue serait alors proportionnelle à la fitness de cet individu comme pour notre segment. La figure 4.2 donne une petite illustration de cette méthode.

Individu	Fitness	Fitness relative
1	15	0.1
2	30	0.2
3	45	0.3
4	22.5	0.15
5	7.5	0.05
6	30	0.2

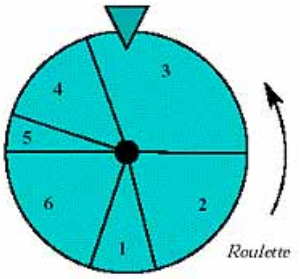

FIGURE 4.2 – Illustration de la méthode de la roulette. Chaque individu est représenté par une case de la roulette. La longueur de cette case est proportionnelle à la fitness de l'individu. Il y a plus de chance de tomber sur un individu si sa case, et donc sa fitness, est grande. Dans cet exemple, l'individu 3 a la fitness la plus grande tandis que l'individu 5 a la plus petite. Source : [20].

Cette méthode de sélection est très simple mais présente toutefois quelques inconvénients. D'une part, dans le cas de petites populations, le tirage risque

de ne pas être représentatif et peut entraîner la perte de bons individus. Il faut donc éviter de prendre de trop petites populations lors de l'utilisation de cette méthode de sélection. D'autre part, ce processus de sélection est très sensible à la variance de la fitness. Si la variance est petite, la sélection par roulette revient presque à un tirage aléatoire tandis que si la variance est trop grande, cela peut entraîner une sélection trop importante des meilleurs individus et la disparition d'individus intéressants. Pour éviter ce genre d'inconvénients, on utilise le scaling et le sharing. Le scaling permet de réduire ou d'augmenter l'écart de fitness entre les individus tandis que le sharing pénalise les fitness communes à un grand nombre d'individus.

Stochastic remainder without replacement (SRWR [15] pp.121-123)

Dans cette méthode, une partie de la population est choisie de façon purement déterministe.

Pour chaque individu, nous devons réaliser les étapes suivantes :

- Calculer le rapport de la fitness de l'individu par la fitness moyenne de l'ensemble des individus.
- Prendre la partie entière de ce rapport.
- Inclure l'individu dans la reproduction le nombre de fois trouvé au point précédent.

Le reste de la population est choisie à l'aide de la sélection par roulette. Ce processus de sélection hybride est représenté dans la figure 4.3

Individu	Fitness	Rapport	Partie entière
1	15	0.6	0
2	30	1.2	1
3	45	1.8	1
4	22.5	0.9	0
5	7.5	0.3	0
6	30	1.2	1

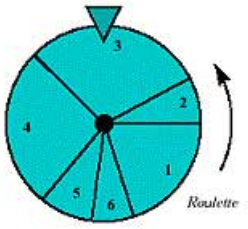

FIGURE 4.3 – Illustration de la méthode SRWR. Certains individus sont choisis de façon déterministe. Dans notre exemple, les individus 2, 3 et 6 sont sûrs d'être sélectionnés au moins une fois. Le reste des individus est sélectionné à l'aide de la sélection par roulette. Source [20].

Le fait de choisir certains individus de façon déterministe élimine partiellement les inconvénients de la sélection par roulette.

Tournoi ([15] p. 121)

Le tournoi est une méthode qui consiste à choisir deux individus à l'aide de la sélection par roulette et à attribuer une probabilité p d'être choisi à celui qui a la plus grande fitness avec $p \in \left(\frac{1}{2}, 1\right]$ fixé. L'individu qui a la plus petite fitness

garde donc une probabilité $1 - p$ d'être choisi. Il existe plusieurs variantes à ce processus. Par exemple, on peut appliquer ce raisonnement à plus de deux individus. Une autre possibilité consiste à utiliser une autre méthode de sélection que la roulette.

Elitisme ([15] pp. 115-118)

Cette stratégie de sélection est souvent couplée à l'une des méthodes vues précédemment. Elle consiste à garder les meilleurs individus de chaque génération afin de les insérer tels quels dans la génération suivante. De cette façon, il n'est pas possible de perdre ces individus lors des procédés aléatoires de sélection ou de mutation.

Il existe plusieurs façons de garder les meilleurs individus. Si nous disposons de k places réservées pour ces individus, nous pouvons choisir de garder les k meilleurs individus ou de choisir k individus parmi les l meilleurs où $l > k$. Nous devons également choisir si la sélection pour le reste de la population se fait sur toute la population ou sur la population sans les individus élites.

4.1.4 Reproduction

La reproduction est, pour rappel, l'opération qui permet le mélange des informations génétiques entre les parents.

La méthode la plus connue pour réaliser cette opération est le croisement ou "crossover". Elle consiste à générer deux individus enfants à partir de deux individus parents en mélangeant les chromosomes de ceux-ci de façon à ce que chacun des enfants possède des gènes de chacun des deux parents. Il existe plusieurs types de crossover et nous allons en détailler quelques-uns fréquemment utilisés.

Crossover à 1 point ([15] p. 12)

Ce crossover, illustré par la figure 4.4, consiste à couper les chromosomes des parents en un point aléatoire et à échanger les fins de chromosomes pour générer les deux enfants. Il est aussi appelé crossover simple.

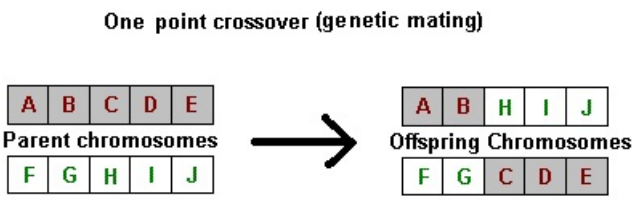

FIGURE 4.4 – Crossover à 1 point. Les chromosomes des parents sont coupés en un point et les fins sont échangées pour donner les enfants. Source : `http://www.softtechdesign.com/GA/EvolvingABetterSolution-GA.html`.

Crossover à n points ([15] pp. 116,119-120)

La figure 4.5 est un exemple de ce type de crossover pour le cas $n = 2$.

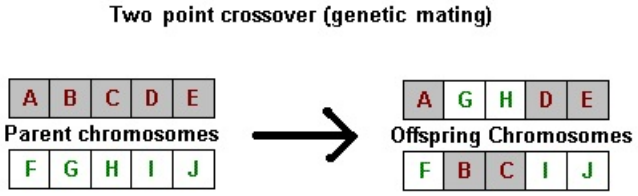

FIGURE 4.5 – Crossover à 2 points. Les chromosomes des parents sont coupés en 2 points et une partie sur deux est échangée entre les parents de façon à générer les enfants. Source : `http://www.softtechdesign.com/GA/EvolvingABetterSolution-GA.html`.

Dans ce cas, n points de coupure sont choisis de façon aléatoire et une partie sur deux des chromosomes est inversée entre les deux parents pour former les enfants.

Crossover à nombres de points variable ([4] p. 19)

Ce type de crossover correspond à un crossover à n points pour lequel n est aléatoire, c'est-à-dire varie d'une utilisation à l'autre.

Crossover uniforme ([4] p. 19)

Lors de ce processus, chaque gène a une probabilité égale à $\frac{1}{2}$ d'être permuté entre les deux parents. Dans la figure 4.6, nous pouvons voir que les gènes A, C et D n'ont pas été permutés tandis que les gènes B et E l'ont été.

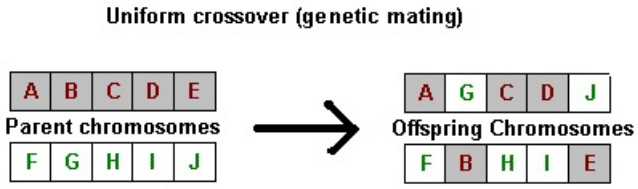

FIGURE 4.6 – Crossover uniforme. Chaque gène a une probabilité égale d'être échangé ou pas entre les deux parents pour générer les enfants. Dans cet exemple, les gènes A, C et D n'ont pas été échangés tandis que les gènes B et E l'ont été. Source : `http://www.softtechdesign.com/GA/EvolvingABetterSolution-GA.html`.

4.1.5 Mutation

Nous avons vu que la mutation consiste à générer de petites modifications dans l'information génétique d'une partie des individus. Ces modifications peuvent apporter une meilleure adaptabilité à l'environnement que celle possédée par

l'individu non muté. Cet opérateur est très important car il empêche une trop grande homogénéité de la population. De nouveau, il existe de nombreuses méthodes pour effectuer cette mutation. Nous allons en donner quelques exemples.

1-inversion ([15] p. 14)

Cette méthode consiste à choisir un gène au hasard et à changer son allèle par une autre valeur possible. La valeur de remplacement est choisie de façon aléatoire.

n-inversion ([4] p. 20)

La n-inversion est semblable à la 1-inversion. En effet, la seule différence est que le processus ne va pas se contenter de changer un gène mais va en changer n.

Inversion totale ([4] p. 20)

Cette inversion correspond à une version extrême des deux précédentes. Dans celle-ci, tous les gènes de l'individu sont modifiés et les allèles sont donc tous remplacés par des valeurs aléatoires.

Après avoir introduit les principales étapes de l'algorithme génétique simple, nous pouvons à présent passer à la description de l'algorithme génétique multi-objectif.

4.2 Algorithmes génétiques multi-objectifs

Dans un grand nombre de problèmes de la vie réelle, nous ne devons pas faire face à un problème d'optimisation d'une seule fonction mais de k fonctions. Par exemple, une entreprise veut maximiser ses revenus tout en minimisant ses coûts de production et de transport. On parle d'optimisation multi-objectif. En général, les différents objectifs à atteindre entrent en conflit les uns avec les autres et il faut donc trouver une solution qui représente un compromis entre ces différents buts.

Il existe différents algorithmes génétiques qui permettent de résoudre de tels problèmes. Parmi ceux-ci, on distingue deux approches principales. La première approche, appelée approche à priori, consiste à combiner les différentes fonctions objectifs pour former une nouvelle fonction objectif et à optimiser cette dernière. La seconde approche, quand à elle, optimise les différents objectifs en utilisant les concepts de dominance et d'optimalité de Pareto. Cette approche est appelée à posteriori. Nous allons à présent détailler ces deux approches. Nous en donnerons également les avantages et inconvénients. Avant cela, nous allons introduire les notions de dominance et d'optimalité de Pareto qui nous permettront de comprendre les différences qui existent entre les deux approches.

4.2.1 Dominance et optimalité de Pareto

Commençons par définir la notion de dominance et de non-dominance pour un individu.

Définition 4.2.1 *Si toutes les fonctions objectifs doivent être minimisées (maximisées), on dit qu'une solution admissible x domine une autre solution admissible y si et seulement si*

- *les valeurs de toutes les fonctions objectifs en x sont inférieures (supérieures) ou égales à celles en y.*
- *il existe au moins une fonction objectif pour laquelle la valeur en x est strictement inférieure (supérieure) à la valeur en y.*

En notations mathématiques, nous aurons :

$$x \succ y \quad \Leftrightarrow \quad \begin{array}{lll} \forall i = 1, \ldots, k & f_i(x) & \leq (\geq) \quad f_i(y) \\ \exists j \in \{1, \ldots, k\} & f_j(x) & < (>) \quad f_j(y) \end{array}$$

Définition 4.2.2 *Une solution admissible est non-dominée s'il n'est pas possible de trouver une autre solution admissible qui la domine.*

La figure 4.7 illustre ces deux notions dans le cas d'une minimisation. Nous pouvons voir que les points A et B sont non-dominés. En effet, on ne peut pas trouver de points admissibles qui soient aussi bons qu'eux dans les deux composantes. Ces deux points dominent très clairement le point C.

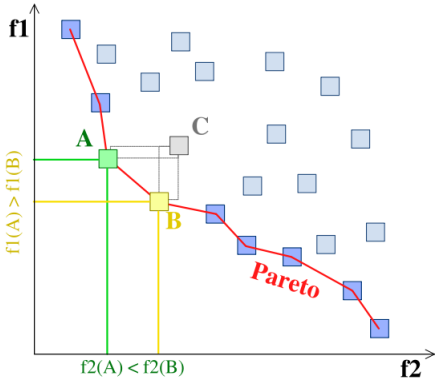

FIGURE 4.7 – Dominance et frontière de Pareto. Les points A et B sont non-dominés. Chacun d'eux domine le point C. La frontière de Pareto est représentée en rouge et correspond à l'ensemble des valeurs des fonctions objectifs pour les solutions admissibles non-dominées. Source : `http://fr.wikipedia.org/wiki/Optimum_de_Pareto`.

Nous pouvons à présent définir la notion de solution Pareto optimale.

Définition 4.2.3 *Une solution admissible est Pareto optimale si elle n'est dominée par aucune autre solution.*

L'ensemble de ces solutions forme un ensemble tout simplement appelé l'ensemble Pareto optimal. Les solutions contenues dans cet ensemble sont bien optimales. En effet, elles sont non-dominées et donc meilleures que les autres pour au moins une fonction objectif. Cela signifie que pour qu'une autre solution soit trouvée, il faut détériorer la valeur de cette fonction de façon à pouvoir en améliorer une autre. Trouver la solution optimale du problème correspond donc

à trouver le meilleur compromis possible entre les différentes fonctions objectifs. A un ensemble Pareto optimal donné, on fait correspondre un ensemble dans l'espace des fonctions, aussi appelé espace objectif. Il est formé des valeurs prises par les fonctions objectifs pour les solutions Pareto optimales. Cet ensemble est appelé la frontière de Pareto (voir figure 4.7).

Après avoir introduit ces nouvelles notions, nous allons pouvoir donner une description plus complète de nos deux approches.

4.2.2 Approche à priori

Comme nous l'avons introduit précédemment, l'approche à priori consiste à combiner les différentes fonctions objectifs de façon à former une nouvelle fonction objectif. L'optimisation se fait ensuite sur la fonction créée en utilisant un algorithme génétique simple.

Pour former cette fonction, les fonctions objectifs sont normalisées si nécessaire. Ensuite, un poids w_i est assigné à chacune d'entre elles. La nouvelle fonction objectif s'écrit alors :

$$f_{new} = w_1 f'_1 + w_2 f'_2 + \cdots + w_k f'_k$$

où f'_i représente la i-ème fonction objectif normalisée et où $\sum_{i=1}^{k} w_i = 1$.

Le terme *à priori* qui caractérise cette approche vient du fait que l'utilisateur est obligé de fournir un poids à chacune des fonctions objectifs. A chaque vecteur de poids assigné à ces fonctions correspond une solution appartenant à l'ensemble des solutions optimales de Pareto. Si l'utilisateur désire connaître toutes les solutions possibles, il doit lancer plusieurs fois son algorithme en modifiant les poids.

Le principal avantage de cette approche est qu'elle utilise un algorithme génétique simple pour optimiser la fonction. Son implémentation est donc assez facile à réaliser puisqu'elle ne nécessite que peu de changements par rapport à un algorithme génétique simple. De plus, ces changements n'interviennent que dans le calcul de la fitness de chaque individu et non pas dans l'algorithme génétique en lui-même. Cependant, cette approche possède un inconvénient assez important. En effet, seule une solution de l'ensemble de Pareto est atteinte à chaque exécution de l'algorithme. De plus, la solution trouvée dépend fortement du vecteur de poids donné par l'utilisateur. Or, celui-ci ne sait pas toujours à l'avance quelle importance il veut donner à chaque critère et il peut passer à côté de solutions intéressantes s'il n'envisage pas assez de vecteurs de poids différents.

4.2.3 Approche à posteriori

Contrairement à l'approche à priori, l'approche à posteriori ne cherche pas à combiner les différentes fonctions objectifs pour former une nouvelle fonction. Au contraire, elle cherche à optimiser directement le problème pour toutes les fonctions objectifs. Pour cela, elle utilise les notions de dominance et de solutions optimales de Pareto. Il existe un grand nombre de méthodes qui ont été développées pour réaliser cette optimisation. Nous ne présenterons ici que celles qui

nous semble intéressantes pour expliquer le développement de ces algorithmes et pour la suite de ce travail.

VEGA [18]

La première méthode à avoir utilisé l'optimisation sur l'ensemble des solutions Pareto optimales est la méthode VEGA (Vector Evaluated Genetic Algorithm). A chaque itération de celle-ci, la population est divisée en k sous-populations. Les individus de chacune de ces sous-populations ont une fitness qui ne dépend que d'une fonction objectif et la sélection à l'intérieur de cette sous-population se fait uniquement selon cette fitness. Les sous-populations sont ensuite remises en commun pour les opérations de crossover et de mutation. Le processus décrit ci-dessus est représenté dans la figure 4.8.

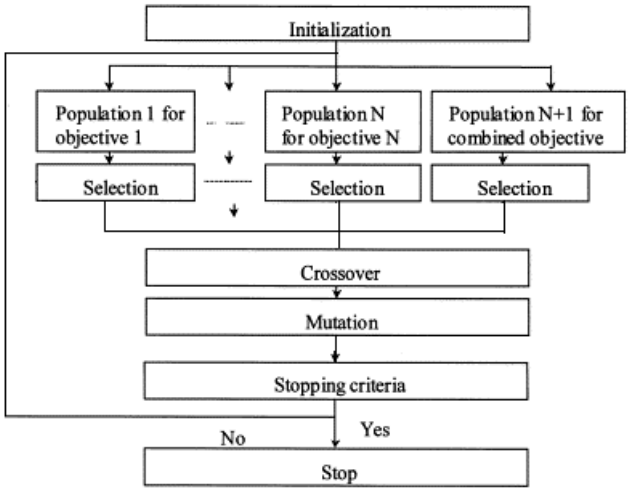

FIGURE 4.8 – Organigramme de la méthode VEGA. A chaque itération, la population est divisée en sous-populations. La fitness de chacune d'elles est calculée en fonction d'un seul objectif et la sélection est effectuée. Les individus sélectionnés dans les sous-populations sont ensuite remis ensemble avant le processus de crossover et de mutation. Le processus est répété à chaque génération. Source : `http://www.sciencedirect.com/science/article/pii/S0957417405000886`.

Cette méthode permet de trouver un ensemble de solutions optimales de Pareto. Elle est toutefois biaisée. En effet, les solutions obtenues ont tendance à être meilleures dans un critère mais assez pauvres dans les autres, ce qui ne représente pas un bon compromis entre les buts à atteindre. Des améliorations peuvent être apportées à cette méthode pour éviter ce genre de biais mais elle devient alors assez complexe.

Tri des points non-dominés : NSGA [9]

Un grand nombre de méthodes telles que MOGA [13], PAES [10], SPEA [18] et NSGA [9] utilise le tri des points non-dominés pour assigner une fitness aux différents individus de leur population. Un rang est tout d'abord donné à chaque

individu en fonction de sa non-dominance par rapport aux fonctions objectifs du problème. Les individus qui sont non-dominés ont un rang de 1 tandis que les individus qui sont juste dominés par les éléments appartenant au rang 1 et qui dominent le reste des individus ont le rang 2 et ainsi de suite. La façon d'assigner la fitness varie ensuite d'une méthode à l'autre. Nous allons présenter ici la façon utilisée dans la méthode NSGA que nous utiliserons par la suite. L'organigramme de cette méthode est présenté dans la figure 4.9

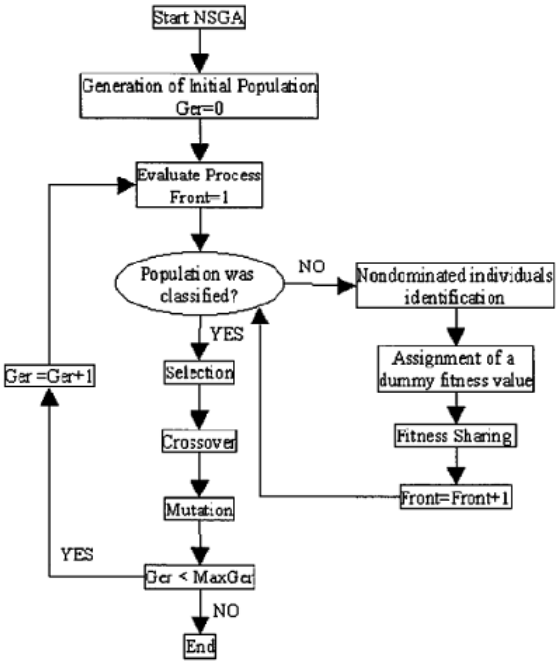

FIGURE 4.9 – Organigramme de la méthode NSGA. Un rang est donné aux individus de la population en fonction de leur non-dominance par rapport aux fonctions objectifs du problème. Une fitness leur est ensuite assignée en fonction de leur rang et de leur distance par rapport aux individus de même rang. Source : [11].

La fitness des individus est calculée rang après rang. Tout d'abord, les individus de rang 1 reçoivent une fitness fictive égale à la taille de la population. Ensuite, une fonction de sharing est utilisée pour pénaliser les individus ayant un grand nombre de voisins de même rang. Une convergence trop rapide vers une seule solution de l'ensemble Pareto optimal est ainsi évitée. Lorsque la vraie fitness des individus de rang 1 a été calculée, c'est-à-dire après l'utilisation de la fonction de sharing, on assigne aux individus de rang 2 une fitness fictive inférieure à la plus petite fitness du rang 1. Le processus de sharing est ensuite relancé pour obtenir les fitness réelles des individus de rang 2 et ainsi de suite.

Lors de la description des méthodes de sélection des algorithmes génétiques classiques, nous avons brièvement expliqué ce qu'était une fonction de sharing. Nous allons à présent donner la façon dont la réduction de la fitness est calculée pour chaque individu.

Supposons que nous avons un ensemble de n individus de même rang. La première étape consiste à calculer la distance existant entre chaque paire d'individus de cet ensemble. Cette distance est donnée par :

$$d_{ij} = \sqrt{\sum_{p=1}^{P} \left(\frac{x_p^{(i)} - x_p^{(j)}}{x_p^u - x_p^l} \right)^2}$$

où $x_p^{(i)}$ est l'allèle du p-ième gène de l'individu i et où P est le nombre total de gènes du chromosome de chaque individu. Les paramètres x_p^u et x_p^l, quand à eux, sont respectivement les bornes supérieures et inférieures de x_p. Ces bornes sont calculées sur tous les individus et non pas simplement sur les individus du rang en cours.

Cette distance est ensuite comparée à un paramètre σ_{share} de façon à calculer la fonction de sharing qui lui est associée :

$$Sh(d_{ij}) = \begin{cases} 1 - \left(\frac{d_{ij}}{\sigma_{share}} \right)^2 & \text{si } d_{ij} \leq \sigma_{share} \\ 0 & \text{sinon} \end{cases}$$

Cette fonction de sharing est également calculée pour chaque paire d'individus. Le "niche count" est alors défini pour chaque individu par :

$$m_i = \sum_{j=1}^{n} Sh(d_{ij})$$

et la fonction de fitness est modifiée de la façon suivante :

$$fit_i \rightarrow \frac{fit_i}{m_i}$$

Cette procédure est évidemment réalisée pour chaque individu du rang en cours.

Nous pouvons faire certaines remarques sur la procédure que nous venons d'expliciter. D'une part, notons que la distance euclidienne calculée dans la première étape est calculée sur l'espace des gènes, aussi appelé espace de décision. Les individus pénalisés sont donc ceux qui ont des allèles similaires. Il est tout à fait possible de prendre une distance dans l'espace objectif. Dans ce cas, les individus pénalisés sont ceux qui ont des valeurs proches pour leurs objectifs. D'autre part, l'utilisation de la fonction de sharing nécessite la connaissance de la valeur du paramètre σ_{share}. Selon Deb [9], une bonne approximation de ce paramètre est la suivante :

$$\sigma_{share} \approx \frac{0.5}{\sqrt[P]{q}}$$

où $q \approx 10$.

Nous allons à présent illustrer la procédure de calcul de la fitness pour une itération de l'algorithme sur un problème test. Ce problème a pour but de minimiser les fonctions $f_1(x) = x^2$ et $f_2(x) = (x-2)^2$. La population est composée

de six individus qui ont un chromosome de longueur 1. Comme $P = 1$, le paramètre σ_{share} est égal à 0.05. L'allèle des individus et leurs valeurs pour les deux fonctions à optimiser sont représentés dans la table 4.1. La dernière colonne de cette table représente leur rang de non-dominance.

	x	f_1	f_2	Rang
1	-1.5	2.25	12.25	2
2	0.7	0.49	1.69	1
3	4.2	17.64	4.84	2
4	2	4	0	1
5	1.75	3.0625	0.0625	1
6	-3.0	9	25	3

TABLE 4.1 – Données d'une itération de l'algorithme sur un problème test. La population est composée de six individus dont les allèles sont repris dans la deuxième colonne. Leurs valeurs pour les deux fonctions objectifs à minimiser sont données dans les colonnes 3 et 4. La dernière colonne contient leur rang de non-dominance.

La première étape consiste à classifier les individus selon leur rang de non-dominance. Dans notre exemple, nous avons trois rangs. Les individus 2, 4 et 5 sont de rang 1 tandis que le 1 et le 3 appartiennent au deuxième rang et que l'individu 6 est de rang 3. Ce classement est évident si nous plaçons nos six individus dans l'espace des objectifs (voir figure 4.10).

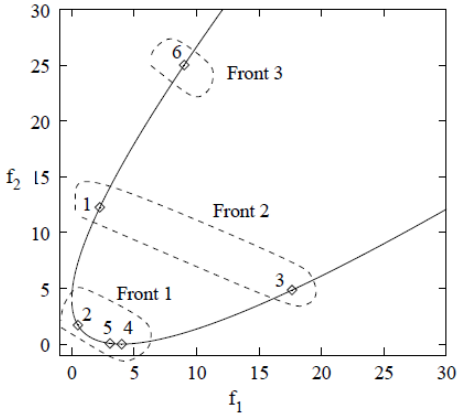

FIGURE 4.10 – Représentation des individus dans l'espace objectif. Les individus 2, 4 et 5 sont non-dominés. Ils sont de rang 1. Les individus 1 et 3 sont dominés par les individus de rang 1. Ils sont de rang 2. L'individu 6, quant à lui, est dominé par tous les autres. Il est donc de rang 3. Source : [9].

La fitness est ensuite calculée rang par rang. Nous allons donc commencer par nous focaliser sur les individus de rang 1, à savoir les individus 2, 4 et 5. Nous leur assignons tout d'abord une fitness fictive égale à la taille de la population, c'est-à-dire une fitness égale à 6. Nous devons ensuite appliquer le processus de sharing pour obtenir la fitness finale.

Nous commençons par calculer la distance entre nos individus de rang 1. Par exemple, la distance entre l'individu 2 et 4 est calculée par :

$$\begin{aligned} d_{24} &= \sqrt{\left(\frac{0.7-2.0}{4.2-(-3.0)}\right)^2} \\ &= \sqrt{\left(\frac{-1.3}{7.2}\right)^2} \\ &= 0.181 \end{aligned}$$

La matrice D de l'équation 4.1 représente la matrice des distances entre nos trois individus. De façon logique, cette matrice est symétrique et composée de zéros sur sa diagonale.

$$D = \begin{pmatrix} 0 & 0.181 & 0.146 \\ 0.181 & 0 & 0.035 \\ 0.146 & 0.035 & 0 \end{pmatrix} \tag{4.1}$$

Nous calculons ensuite la fonction de sharing associée à chacune de ces distances. Pour rappel, le paramètre σ_{share} est égal à 0.05. Nous obtenons la matrice Sh (voir équation 4.2).

$$Sh = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.518 \\ 0 & 0.518 & 1 \end{pmatrix} \tag{4.2}$$

Nous pouvons maintenant calculer le "niche count" de chaque individu.

$$\begin{aligned} m_1 &= Sh_{11} + Sh_{12} + Sh_{13} \\ &= 1 \\ m_2 &= 1.518 \\ m_3 &= 1.518 \end{aligned}$$

Nous terminons cette procédure en modifiant les fitness fictives fit' afin d'obtenir les fitness réelles fit.

$$\begin{aligned} fit_1 &= \frac{fit'_1}{m_1} \\ &= 6 \\ fit_2 &= 3.95 \\ fit_3 &= 3.95 \end{aligned}$$

Nous avons assigné une fitness aux individus de rang 1. Nous devons encore faire de même pour les individus de rang 2 et de rang 3. Nous assignons une fitness fictive égale à 3.5 aux individus de rang 2. Conformément à la théorie, cette valeur est plus petite que la fitness minimale des individus de rang 1. La distance entre les deux individus de rang 2 est de 0.792. Comme cette distance est supérieure au paramètre σ_{share} (= 0.05), les fitness réelles seront égales aux fitness fictives. Nous assignons ensuite une fitness fictive égale à 3.0 à l'individu

de rang 3. Comme cet individu est seul, il ne peut pas être proche d'un autre et nous ne devons pas appliquer la procédure de sharing. La fitness réelle est donc égale à 3.0 et nous avons terminé le calcul des fitness des individus de notre exemple. La procédure de sélection de l'algorithme génétique peut alors être lancée sur les fitness réelles. Le tableau 4.2 reprend les fitness fictives et réelles de nos six individus.

	Fitness fictive fit'	Fitness fit
1	3.5	3.5
2	6.0	6.0
3	3.5	3.5
4	6.0	3.95
5	6.0	3.95
6	3.0	3.0

TABLE 4.2 – Fitness fictives et réelles obtenues pour les six individus.

Remarque
Dans cet exemple, nous avons pris une différence assez grande (environ 0.5) entre la fitness minimale des individus d'un rang et la fitness fictive des individus du rang suivant. Nous avons choisi cette valeur pour la différence de façon à illustrer le passage d'un rang à un autre. En général, la valeur prise est plus petite (0.1 ou 0.05), ce qui permet de ne pas éliminer trop rapidement des individus intéressants.

Le principal avantage des méthodes utilisant le tri des points non-dominés est de converger vers un ensemble de Pareto proche du réel. Cela permet à l'utilisateur de comparer les différentes solutions avant de choisir celle qui lui convient en fonction de ses priorités. Le nom d'approche *à posteriori* découle évidemment de ce fait. Ces méthodes ont toutefois certains désavantages. En effet, le temps d'exécution de l'algorithme est plus long car celui-ci doit trier les points non-dominés et doit également effectuer la procédure de sharing. Les méthodes de sélection, de reproduction et de mutation restent les mêmes que dans les algorithmes génétiques classiques.

Mesures de performance

Il existe plusieurs mesures permettant de calculer le niveau de performance d'un algorithme génétique multi-objectif à posteriori. En effet, ceux-ci n'ont pas simplement pour but de trouver une solution mais bien un ensemble de solutions Pareto optimales qui soit le plus proche possible du réel. Ces mesures sont calculées à chaque itération de façon à vérifier si le comportement observé est bien celui attendu. Nous allons présenter trois de ces mesures.

La première mesure est le nombre de points Pareto optimaux différents qui sont atteints à chaque itération. Cela permet de voir si certains d'entre-eux ne sont pas perdus en route à cause d'une fitness trop petite ou d'une convergence trop rapide vers un autre point optimal. Il est toutefois important de signaler que cette mesure n'est pas très précise. En effet, le fait d'avoir un grand nombre

de points Pareto optimaux ne signifie nullement que ceux-ci soient proches des vraies solutions Pareto optimales.

La deuxième mesure, appelée espacement (spacing en anglais), est donnée par l'équation suivante :

$$S = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\langle distm\rangle - distm_i)^2}$$

où

- n est le nombre de solutions Pareto optimales différentes,
- $distm_i$ est la distance de Manhattan minimale entre la solution x_i et toutes les autres solutions, c'est à dire

$$distm_i = \min_{\substack{j=1\ldots n \\ j\neq i}} \sum_{m=1}^{k} |f_m(x_i) - f_m(x_j)|$$

- $\langle distm\rangle$ est la moyenne des $distm_i$.

Cette mesure représente la déviation standard de la distance minimale moyenne entre deux solutions. Une valeur proche de 0 pour cet indice indique que les solutions sont distribuées uniformément le long de la frontière de Pareto trouvée par l'algorithme. Cette mesure n'est toutefois pas suffisante pour déterminer si l'algorithme fonctionne correctement. En effet, les solutions peuvent être distribuées uniformément le long des deux extrémités et ne représenter que les solutions extrêmes du front optimal.

La dernière mesure que nous introduisons est appelée mesure d'hétérogénéité. Sa définition est :

$$H = \frac{1}{n}\sum_{i=1}^{n} dist_i$$

où

- n représente de nouveau le nombre de solutions Pareto optimales différentes,
- $dist_i$ est la distance euclidienne moyenne entre la solution x_i et toutes les autres solutions.

Un algorithme génétique multi-objectif efficace aura une valeur importante pour cette mesure. En effet, si H est grand, cela signifie que la distance moyenne entre les solutions est grande et donc que nos points sont éloignés les uns des autres.

Si un algorithme génétique multi-objectif donne des résultats satisfaisants pour ces trois mesures, cela signifie qu'il trouve suffisamment de solutions Pareto optimales, que ces solutions sont distribuées régulièrement le long de la frontière de Pareto et qu'elles ne sont pas trop proches les unes des autres. Dans ce cas, on peut alors conclure que l'algorithme génétique multi-objectif fonctionne correctement.

4.3 Apprentissage des réseaux

Après avoir introduit le concept d'algorithmes génétiques (simples et multi-objectifs) et détaillé leurs différentes phases, nous allons à présent les utiliser dans l'apprentissage de nos réseaux de neurones. Nous commencerons par expliquer pourquoi ce type d'algorithmes est très utile pour cet apprentissage. Nous donnerons ensuite quelques indications pratiques sur la façon dont nous avons implémenté les réseaux et les algorithmes génétiques. Nous terminerons en illustrant leur utilisation sur une fonction logique simple.

Nous avons vu dans le chapitre précédent que les méthodes classiques d'apprentissage permettent de trouver les poids du réseau afin que celui-ci réalise une tâche particulière. Les algorithmes génétiques, quant à eux, permettent de travailler sur différents aspects de nos réseaux. En effet, nous pouvons les utiliser non seulement pour trouver les poids de nos réseaux mais aussi pour optimiser leur architecture ou pour optimiser les paramètres des méthodes classiques d'apprentissage. Or, dans la suite de ce travail, nous allons autant nous intéresser à la recherche des poids permettant à un réseau de réaliser parfaitement la tâche qu'à l'optimisation de l'architecture de ce réseau. Nous avons donc choisi d'implémenter un algorithme génétique plutôt qu'une méthode classique d'apprentissage pour faire apprendre nos réseaux.

Remarque
Il existe des méthodes classiques d'apprentissage qui permettent également d'optimiser l'architecture des réseaux ou d'autres paramètres. Toutefois, elles sont assez complexes et ne concernent pas les réseaux construits sur les modèles de Mc-Culloch-Pitts et du perceptron.

4.3.1 Implémentation

A présent que nous avons donné la raison pour laquelle nous avons choisi d'utiliser un algorithme génétique comme mécanisme d'apprentissage, nous allons donner quelques indications sur la façon dont nous avons implémenté nos réseaux de neurones et nos algorithes génétiques.

Réseaux de neurones

Les réseaux de neurones que nous avons implémentés sont des réseaux de la forme du perceptron qui ont été légèrement simplifiés. En effet, les poids ne sont plus des valeurs réelles comprises entre -1 et 1 mais peuvent valoir uniquement 0 (pas de connexion entre les deux neurones concernés), 1 (excitation) ou -1 (inhibition). Un neurone ne peut pas être en connexion avec lui-même. Les seuils, quant à eux, sont compris dans l'intervalle $[-1, 1]$. La règle d'activation de chaque neurone reste celle du perceptron.

Ces réseaux sont représentés par une matrice d'adjacence avec poids W dont les éléments vérifient la relation :

$$W_{ij} \neq 0 \Leftrightarrow v_j \rightarrow v_i$$

Par exemple, si nous prenons la matrice

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 \\ 1 & 1 & -1 & 0 \end{pmatrix} \tag{4.3}$$

qui représente le réseau à quatre neurones de la figure 4.11, nous pouvons voir qu'une connexion lie le neurone 1 au neurone 4 puisque W_{41} est différent de zéro. Par la suite, nous écrirons les seuils des neurones sur la diagonale de cette matrice. Puisque nous avons posé qu'un neurone ne peut pas être en connexion avec lui-même, il n'y aura pas de confusion entre les poids et les seuils. De plus, cette notation nous permettra d'avoir toute la description du réseau de neurones dans la matrice.

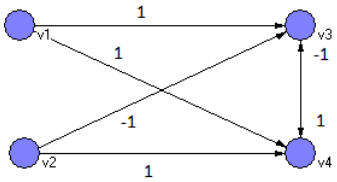

FIGURE 4.11 – Exemple de réseaux de neurones. Ce réseau a été construit à partir de la matrice présentée dans l'équation 4.3. Si l'élément W_{ij} de cette matrice est différent de zéro, cela signifie qu'il existe une connexion qui va du neurone j au neurone i. Si l'arc est double, le poids de la connexion se trouve du côté du neurone dont part la connexion.

La valeur d'un neurone est initialisée à 0 pour les neurones qui ne représentent pas les entrées. La valeur de ces entrées, quant à elle, ne peut jamais être modifiée au cours de l'exécution. Lors de l'exécution du réseau, nous ne nous contentons pas d'une évaluation de celui-ci. En effet, il peut exister un temps de transition avant que le réseau ne se stabilise en un état d'équilibre ou en un cycle périodique d'états. Nous devons donc atteindre cette stabilisation avant de pouvoir observer les sorties de notre réseau.

Remarque
Une réseau de neurones aura toujours une orbite périodique. En effet, il s'agit d'un système discret, c'est-à-dire un système avec un nombre fini d'états possibles. La période peut cependant être assez grande (environ 2^n au maximum).

La figure 4.12 illustre deux exécutions complètes de réseaux de 6 neurones choisis aléatoirement. Les carrés noirs sont les neurones qui ont une valeur de 0 et les blancs sont ceux qui ont une valeur de 1. Le graphe de gauche représente une exécution pour laquelle la période de transition est égale à trois pas de temps et pour laquelle l'état du réseau se stabilise en un état d'équilibre. Pour le graphe de droite, la période de transition est égale à un pas de temps et le réseau se stabilise en un cycle périodique d'états de période quatre. Dans ces

deux graphes, les valeurs des deux premiers neurones ne sont jamais modifiées car ceux-ci correspondent aux entrées du réseau.

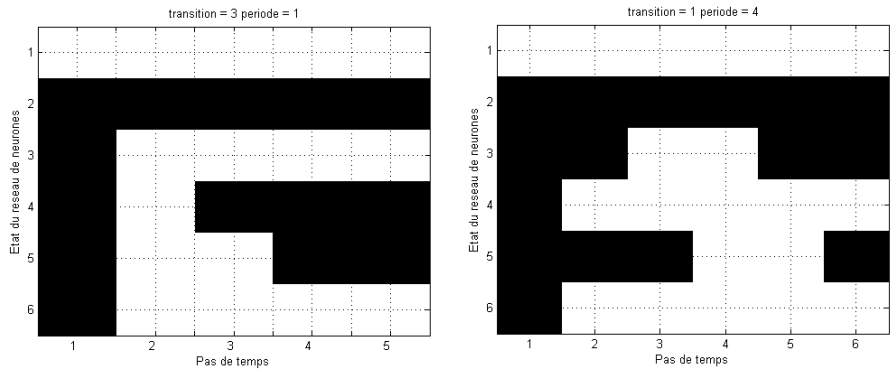

FIGURE 4.12 – Valeurs des neurones du réseau à chaque pas de temps. Les carrés noirs représentent les neurones dont la valeur est 0 tandis que les blancs correspondent aux valeurs égales à 1. Dans le graphe de gauche, la période de transition de l'exécution est égale à trois pas de temps et le réseau se stabilise en un état d'équilibre. Pour le graphe de droite, la période de transition est égale à un pas de temps et le réseau se stabilise en un cycle périodique d'états de période quatre.

Algorithme génétique

Notre algorithme génétique représente une méthode d'apprentissage supervisée. En effet, pour chaque apprentissage, nous allons devoir fournir aux réseaux un ensemble de combinaisons d'entrées couplé à un ensemble de cibles.

Pour apprendre une tâche à nos réseaux, nous allons devoir utiliser un algorithme génétique multi-objectif. En effet, nous cherchons à obtenir des réseaux qui réalisent une tâche donnée. Mais nous souhaitons également que ce réseau possède un minimum de connexions et qu'il réalise la tâche en un minimum de temps. En d'autres mots, nous voulons optimiser les poids du réseau de façon à ce qu'il réalise la tâche voulue, son architecture pour qu'il ne possède pas de liens inutiles et son temps d'exécution. Nous allons donc avoir trois fonctions objectifs qui sont détaillées ci-dessous.

- Objectif 1 : la pénalité sur le nombre d'erreurs commises par rapport à la cible. Pour obtenir la valeur de ce premier objectif, nous exécutons tout d'abord une simulation de notre réseau de neurones afin d'atteindre le cycle périodique d'états (ou l'état d'équilibre). Nous comparons ensuite les sorties des états finaux, c'est-à-dire les sorties des états appartenant au cycle périodique, aux sorties attendues. Cette simulation et cette comparaison sont effectuées pour les différentes combinaisons d'entrées dont nous disposons. Lorsque le réseau a commis des erreurs, il est pénalisé. L'équation de ce premier objectif est la suivante :

$$1 - \sum_{i=1}^{c} \sum_{j=1}^{T_i} \frac{\epsilon_{ij}}{\epsilon_{max}}$$

où

- c représente le nombre de combinaisons d'entrées,
- T_i est la période du réseau pour la combinaison d'entrées i,
- ϵ_{ij} représente l'erreur commise au temps j de la période pour la combinaison d'entrées i. Cette erreur correspond au nombre de sorties de l'état du réseau au temps j qui diffèrent des sorties attendues pour la combinaison d'entrées i,
- ϵ_{max} est le nombre maximal d'erreurs qui peuvent être commises, c'est-à-dire $\epsilon_{max} = \sum_{i=1}^{c} nb\_sorties \times T_i$.

- Objectif 2 : la pénalité imputée aux réseaux denses. Cette pénalité est calculée en regardant le nombre de connexions du réseau par rapport au nombre total de connexions possibles. L'équation de cette pénalité est

$$1 - \frac{l}{n \times (n-1)}$$

où l est le nombre de connexions (ou liens) du réseau et où n est le nombre de neurones du réseau. La valeur $n \times (n-1)$ correspond alors au nombre maximum de connexions qui peuvent relier les neurones du réseau entre-eux. Pour rappel, un réseau ne peut pas être relié à lui-même d'où la valeur de $n \times (n-1)$ et pas n^2.

- Objectif 3 : la pénalité infligée aux réseaux lents. Pour calculer cette pénalité, nous additionnons la période et le temps de transition du réseau. Nous comparons ensuite ce résultat avec le nombre d'états possibles. Rappelons que la valeur des neurones d'entrées est fixée. Nous ne devons donc pas en tenir compte quand nous calculons le nombre d'états possibles. Nous répétons l'opération pour chaque combinaison d'entrées possibles. L'équation de ce dernier objectif est donnée par

$$1 - \frac{1}{c} \sum_{j=1}^{c} \frac{(trans_j + periode_j)}{2^{n - nb\_entrées}}$$

où c représente de nouveau le nombre de combinaison d'entrées.

Chacun de ces objectifs est compris entre 0 et 1 et nous devons maximiser chacun d'eux. En effet, un réseau qui ne commet pas d'erreur par rapport à la cible aura une valeur de 1 pour le premier objectif par exemple.

Comme nous l'avons vu dans la section précédente, il existe deux types d'algorithmes génétiques multi-objectifs. Nous allons implémenter une version de chacun de ces types. Pour le type à priori, nous allons définir une fonction de fitness à partir des trois objectifs à optimiser. Pour cela, nous allons devoir trouver une combinaison de poids qui nous donne des résultats satisfaisants. Pour le type à posteriori, nous allons utiliser le concept d'optimalité de Pareto et la fonction de sharing développés précédemment pour définir la fonction de fitness. Notre algorithme sera donc un algorithme de type NSGA.

Le principe général de ces deux algorithmes est le même. Nous prenons une population aléatoire de réseaux de taille donnée que nous faisons évoluer pendant un certain nombre de générations afin d'obtenir le meilleur réseau possible. Nous utilisons une sélection par roulette. Pour la reproduction, nous choisissons aléatoirement une colonne de la matrice et nous échangeons les colonnes qui suivent entre les deux réseaux parents pour obtenir les enfants. Ce processus est un crossover à 1 point. La mutation utilisée, quant à elle, est une 1-inversion qui est appliquée sur un réseau au hasard. Le nombre de crossover et de mutation à effectuer est donné par leur taux respectif. Pour éviter une convergence trop rapide vers un individu, nous introduisons de nouveaux individus aléatoires à chaque génération.

Avant de terminer ce chapitre, nous allons illustrer l'utilisation de nos deux algorithmes sur une fonction logique simple : la fonction *ou*.

4.3.2 Exemple d'utilisation : la fonction *ou*

Nous allons tenter de trouver le réseau optimal qui modélise la fonction *ou*. Nous avons vu lorsque nous avons analysé nos réseaux que cette fonction pouvait être modélisée par trois neurones représentant les deux entrées et la sortie. Comme notre objectif est simplement d'illustrer l'utilisation de nos algorithmes, nous allons prendre des réseaux de neurones aléatoires de taille 3 et nous allons les faire évoluer. Nous vérifierons ensuite que nous obtenons bien le résultat voulu. Nous réaliserons cette analyse pour nos deux algorithmes. Par la suite, nous pourrons prendre des réseaux de tailles différentes et voir ce qui se passe lors de nos exécutions.

Algorithme génétique à fitness pondérée

L'analyse des résultats obtenus pour la fonction logique *ou* avec cet algorithme va nous permettre d'illustrer l'importance du poids accordé à chaque fonction objectif. En effet, nous allons voir que suivant les poids donnés, le réseau vers lequel l'algorithme converge n'est pas le même. Pour cet algorithme, nous considérons que le meilleur réseau est l'individu de la population finale qui possède la plus grande fitness.

Commençons par prendre une fonction de fitness qui ne tient compte que du premier objectif, à savoir la pénalité sur le nombre d'erreurs commises par rapport à la cible. Dans ce cas, la matrice représentant le meilleur réseau obtenu à la fin de notre exécution est la suivante :

$$\begin{pmatrix} 0 & 0 & -1 \\ -1 & 0 & -1 \\ 1 & 1 & 0.25 \end{pmatrix} \tag{4.4}$$

Si nous regardons les états finaux de ce réseau, nous pouvons voir que celui-ci modélise bien la fonction *ou*. En effet, si nous calculons la sortie en fonction des différentes combinaisons d'entrées possibles, nous obtenons à chaque fois la sortie attendue. Cependant, si nous analysons une représentation graphique de ce réseau (voir figure 4.13), nous pouvons voir que certaines connexions sont

inutiles. Par exemple, les valeurs des neurones d'entrées ne peuvent pas être modifiées au cours de l'exécution. Les connexions agissant sur ces entrées sont donc inutiles et il est possible de trouver de meilleurs réseaux qui modélisent la fonction. Le réseau obtenu n'est donc pas optimal.

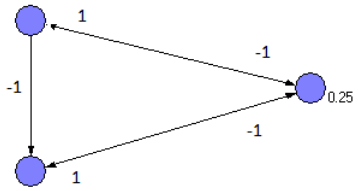

FIGURE 4.13 – Réseau obtenu avec une fitness uniquement basée sur le premier objectif. Ce réseau est représenté par la matrice de l'équation 4.4. Les deux neurones de gauche sont les entrées du réseau tandis que celui de droite est la sortie. Il n'y a pas de neurones intermédiaires. Certaines connexions sont inutiles. Le réseau n'est donc pas optimal pour la tâche à réaliser.

Remarque
La solution de ce problème n'est pas unique. En effet, le réseau observé ne représente qu'une des solutions possibles si nous ne tenons compte que du premier objectif. D'une part, nous pourrions trouver d'autres réseaux capables de modéliser la fonction *ou* et possédant d'autres connexions inutiles. D'autre part, même si nous gardons les mêmes connexions, le seuil du neurone de sortie peut varier. Nous obtiendrons toujours la modélisation voulue si nous prenons une seuil de 0.2 ou 0.3 à la place du seuil de 0.25 utilisé dans le réseau de la figure 4.13. La solution obtenue va varier d'une exécution à l'autre de l'algorithme.

Remplaçons à présent la fitness utilisée ci-dessus par une combinaison tenant compte des deux premiers objectifs, c'est-à-dire la pénalité sur les erreurs commises par rapport à la cible et la pénalité imputée aux réseaux plus denses. Le poids donné à chacun de nos deux objectifs est de 0.8 pour le premier et 0.2 pour le second. La matrice du meilleur réseau obtenu à l'aide de notre algorithme est alors :

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0.44 \end{pmatrix} \tag{4.5}$$

où, pour rappel, les seuils des neurones sont situés sur la diagonale. Cette matrice correspond au réseau de neurones représentés dans la figure 4.14

Nous pouvons voir que les connexions inutiles ont disparu, ce qui signifie que l'introduction de la pénalité sur le nombre de connexions des réseaux permet d'améliorer le réseau final que nous obtenons. De plus, si nous comparons la valeur du troisième objectif pour le meilleur réseau de nos deux exécutions de l'algorithme génétique, nous pouvons voir que celle-ci est la même et nous n'avons donc pas ralenti l'exécution de notre réseau en supprimant les connexions inutiles. Signalons toutefois qu'il ne faut pas donner trop d'importance à la pénalité sur le

nombre de connexions. En effet, si nous lui donnons un coefficient de 0.5 (et 0.5 pour le premier objectif), le meilleur réseau final ne contient plus aucune connexion. Dans ce cas, l'objectif 2 est bien maximisé mais le réseau ne modélise plus la fonction voulue.

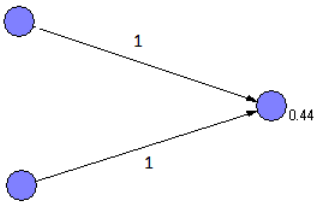

FIGURE 4.14 – Réseau obtenu avec une fitness basée sur les deux premiers objectifs. Ce réseau est représenté dans l'équation 4.5. Les deux neurones de gauche sont toujours les entrées du réseau tandis que celui de droite est la sortie. Il n'y a pas de neurones intermédiaires. Les connexions inutiles ont disparu. Le réseau est optimal.

Remarque
Si nous utilisons une combinaison des deux premiers objectifs dans notre fitness, nous faisons disparaître les connexions inutiles. Nos réseaux optimaux auront donc toujours les mêmes connexions d'une exécution à l'autre de l'algorithme. Cette fois, seul le seuil du neurone de sortie peut varier. En effet, les seuils des neurones d'entrées sont fixés puisque la valeur des entrées ne peut pas changer.

Continuons notre exemple en ajoutant cette fois le troisième objectif, c'est-à-dire la pénalité sur le temps d'exécution, aux deux précédents. Les poids donnés à nos trois objectifs sont respectivement de 0.6, 0.2 et 0.2. Dans ce cas, nous obtenons le même meilleur réseau final que précédemment. Cela signifie que l'ajout du troisième critère dans notre exemple ne change pas le résultat. Cependant, nous travaillons sur un réseau de très petite taille et nous verrons par la suite si ce critère prend de l'importance dans des réseaux plus grands. De nouveau, le coefficient donné à cette pénalité sur le temps d'exécution ne doit pas être trop important, au risque de perdre l'objectif premier qu'est la modélisation de la fonction.

Nous venons de voir que notre algorithme génétique à fitness pondérée nous permet de converger vers le réseau optimal à condition d'assigner un poids adéquat à chacun de nos objectifs. Par la suite, nous analyserons plus longuement notre algorithme et nous tenterons de trouver la meilleure combinaison de poids possibles.

Algorithme génétique multi-objectif NSGA

Contrairement à l'algorithme précédent qui converge vers une seule solution optimale en fonction du vecteur de poids qu'on lui assigne, l'algorithme génétique multi-objectif NSGA converge vers un ensemble de solutions Pareto optimales. Nous devons donc choisir la solution qui nous convient à la fin de l'exécution.

Notre premier objectif étant de trouver un réseau optimal qui modélise correctement la fonction voulue, nous allons nous intéresser uniquement aux solutions Pareto optimales qui ont une valeur proche ou égale à 1 pour le premier objectif. Si cette valeur n'apparaît pour aucune de nos solutions, cela signifie que notre population n'a pas convergé vers une solution optimale adéquate.

Si nous analysons les résultats obtenus pour la fonction *ou* modélisée par des réseaux de trois neurones, nous obtenons bien une solution Pareto optimale qui a une valeur de 1 pour le premier objectif. De plus, la matrice de cette solution a la même structure que celle de l'équation 4.5 qui représente le réseau de la figure 4.14. Nous obtenons donc bien la même solution optimale qu'avec l'algorithme génétique avec poids.

Le fait d'obtenir la même solution que pour l'algorithme précédent n'est pas suffisant pour affirmer que notre algorithme génétique NSGA fonctionne correctement. En effet, nous avons vu que ce type d'algorithmes doit donner des solutions Pareto optimales différentes et que ces solutions doivent être suffisamment hétérogènes et uniformément distribuées. Pour vérifier ces critères, nous avons implémenté les trois mesures de performance décrites précédemment, à savoir le nombre de solutions Pareto optimales différentes, l'espacement et l'hétérogénéité.

Si nous calculons ces mesures pour notre exemple, nous obtenons 3 ou 4 solutions non-dominées à chaque étape. Celles-ci ne sont évidemment pas les mêmes tout au long de l'algorithme. En effet, la découverte d'un nouveau point possédant une meilleure valeur pour un objectif entraîne l'abandon de certains points qui appartenaient à l'ensemble optimal et ainsi de suite. Les résulats obtenus pour les mesures d'espacement et d'hétérogénéité sont représentés dans la figure 4.15.

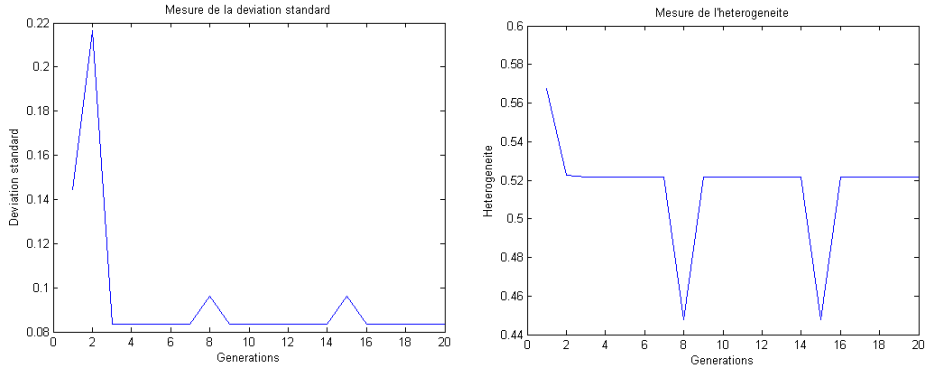

FIGURE 4.15 – Mesures de déviation standard et d'hétérogénéité. Ces mesures sont calculées pour chaque génération de l'algorithme. La figure de gauche représente l'espacement et celle de droite l'hétérogénéité. L'espacement est assez proche de 0. Les solutions sont donc identiquement distribuées. L'hétérogénéité se stabilise assez rapidement, ce qui signifie que le spectre des solutions optimales ne diminue pas au cours de l'algorithme.

Pour l'espacement (graphe de gauche), nous obtenons une valeur assez proche de 0 pour les dernières générations ce qui, comme nous l'avons vu, signifie que les solutions sont uniformément distribuées. La mesure d'hétérogénéité, quant à elle, a une valeur qui décroît dans les premières générations avec l'apparition de nouvelles solutions avant de se stabiliser (voir figure de droite). Cela signifie que nos solutions ne sont pas très éloignées les unes des autres mais nous ne réduisons toutefois pas le spectre des solutions optimales au cours de notre algorithme. Dans ces deux graphes, nous pouvons voir deux pics aux générations 8 et 15. Ces pics correspondent à la perte d'une solution qui est toutefois directement récupérée à l'itération suivante.

Remarque
Nous ne nous sommes pas contentés de cette exemple pour décider de l'efficacité de notre algorithme. En effet, notre exemple représente un problème assez simple qui ne possède pas beaucoup de solutions optimales. Nous avons donc également réalisé cette analyse des mesures de l'algorithme pour l'optimisation d'autres tâches. Comme les résultats obtenus pour les différentes optimisations sont assez bons, nous concluons que l'algorithme que nous avons implémenté est efficace. Nous allons donc pouvoir passer à des analyses plus précises sur les paramètres utilisés par celui-ci. Nous utiliserons également nos algorithmes génétiques pour analyser le comportement de nos réseaux optimaux.

4.4 Conclusion

Dans ce chapitre, nous avons étudié les algorithmes génétiques simples et multi-objectifs de façon théorique. Nous avons également mis en avant l'avantage d'utiliser ces algorithmes pour faire évoluer nos réseaux. Nous avons ensuite implémenté un modèle de réseaux de neurones et deux algorithmes génétiques multi-objectifs : un suivant l'approche à priori et l'autre à posteriori. Nous avons donné quelques explications concernant la façon dont ces différents éléments ont été implémentés. Nous avons terminé ce chapitre en vérifiant le bon fonctionnement de nos algorithmes sur un exemple simple.

Nous avons montré que nos algorithmes fonctionnent correctement à condition de choisir de bonnes valeurs pour leurs paramètres. Le chapitre suivant va donc avoir pour objectif d'analyser ces paramètres de façon plus précise afin d'obtenir les algorithmes les plus efficaces possibles. Nous pourrons ensuite passer à l'analyse des réseaux optimaux obtenus lors des exécutions de nos algorithmes.

Chapitre 5

Analyse des paramètres et des réseaux optimaux

Ce chapitre a pour but d'analyser en détail nos algorithmes et les réseaux optimaux qu'ils nous fournissent. Pour cela, nous allons tout d'abord analyser les résultats obtenus en fonction des valeurs introduites pour chacun des paramètres de nos algorithmes génétiques. L'objectif de cette analyse est de trouver la combinaison de paramètres qui nous permet de converger vers le réseau optimal et ce le plus rapidement possible, c'est-à-dire en un nombre minimum de générations. Une fois que nous l'aurons trouvée, nous étudierons la dégénérescence et la redondance des réseaux obtenus à l'aide de nos algorithmes. Nous chercherons également des réponses à certaines questions dont une qui nous intéresse particulièrement : Y a-t-il une zone du cerveau réservée à chaque tâche ou certaines d'entre-elles sont-elles réalisées par les mêmes neurones ?

Ce chapitre est l'aboutissement d'une recherche personnelle basée sur les connaissances théoriques acquises lors des chapitres précédents. Les codes ayant servi à l'élaboration des résultats sont fournis dans les annexes. Une partie de ce chapitre a été réalisée au ECLT (European Center for Living Technology) à Venise sous la supervision d'Irene Poli et de Davide de March.

Comme certaines des analyses réalisées au cours de ce chapitre étaient assez longues (exécution de 2 ou 3 jours), nous avons dû nous servir de ressources informatiques autres que celles disponibles sur l'ordinateur d'un particulier ou au pool informatique. Nous avons donc appris à utiliser les ressources fournies par l'ISCF (Interuniversity Scientific Computing Facility). Nous avons également utilisé les ressources fournies à l'ECLT (processeurs plus puissants).

5.1 Analyse des paramètres

Notre but est d'analyser les réseaux de neurones et les algorithmes génétiques en fonction des différents paramètres de notre implémentation. Nous allons tout d'abord chercher les combinaisons de paramètres permettant une convergence rapide vers le réseau optimal. Cette phase préliminaire d'analyse sera suivie d'une seconde phase qui servira à répondre à certaines questions concernant la

façon dont fonctionnent nos réseaux.

Nous commencerons par l'analyse de l'algorithme génétique à fitness pondérée. Nous ferons ensuite de même avec l'algorithme génétique multi-objectif de type NSGA. Nous finirons cette partie en comparant nos deux algorithmes.

Cette analyse sera exécutée sur des réseaux de petite taille qui doivent modéliser des fonctions booléennes relativement simples. Nous allons travailler sur différentes tâches. Nous allons donner un nom à ces tâches de façon à pouvoir les référencier facilement. La première tâche, appelée tâche A, consiste à modéliser la fonction *ou* à l'aide d'un réseau de trois neurones (deux entrées et une sortie). Cette première tâche correspond à l'exemple sur lequel nous avons travaillé lors du chapitre précédent. L'objectif de la tâche B est de faire modéliser les fonctions logiques *ou* et *et* par un réseau de quatre neurones (deux entrées et deux sorties). Il s'agit donc d'une tâche un peu plus difficile à résoudre. Le nombre de générations effectuées lors de l'exécution de ces deux tâches est 50 pour la tâche A et 100 pour la tâche B.

L'avantage concernant ces deux tâches est que nous pouvons trouver le réseau optimal qui modélise la fonction demandée sans avoir recours à nos algorithmes. Les réseaux optimaux des tâches A et B sont représentés dans la figure 5.1. Cela nous permettra de comparer les résultats obtenus lors de nos exécutions avec ce réseau optimal et de voir quelles sont les valeurs des paramètres qui l'atteignent. Nous utiliserons également des réseaux de taille 6 pour modéliser les fonctions présentées dans les tâches A et B. De cette façon, nous pourrons observer ce qui se passe quand le réseau possède des neurones intermédiaires. Pour nos deux problèmes, ceux-ci sont inutiles et n'auront donc pas de connexions si le réseau obtenu est optimal. Ces deux tâches sont appelées tâche C et tâche D. Le nombre de générations par exécution est de 200.

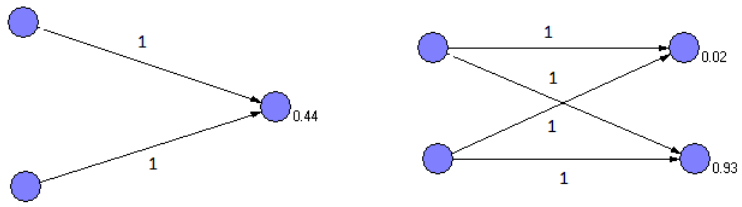

FIGURE 5.1 – Représentation des réseaux optimaux. Le réseau de gauche est composé de trois neurones tandis que celui de droite est composé de quatre neurones. Les fonctions modélisées sont, à gauche, la fonction logique *ou* et à droite les fonctions logiques *ou* et *et*.

Avant de passer à l'analyse proprement dite, nous allons récapituler les différentes tâches sur lesquelles nous allons travailler dans la table 5.1. Ce tableau reprend pour chacune des tâches le nombre de neurones qui composent le réseau, son nombre d'entrées, de sorties et de neurones intermédiaires (ou cachés), la fonction à modéliser et le nombre maximal de générations par exécution.

Tâche	Neurones	Inputs	Outputs	Cachés	Fonction	Nb gen
A	3	2	1	0	ou	50
B	4	2	2	0	ou, et	100
C	6	2	1	3	ou	200
D	6	2	2	2	ou, et	200

TABLE 5.1 – Tableau récapitulatif des différentes tâches utilisées lors de nos analyses.

5.1.1 Algorithme génétique à fitness pondérée

Commençons par analyser notre algorithme génétique à fitness pondérée. Les paramètres sur lesquels nous allons travailler sont les poids des trois objectifs de la fitness ainsi que leur interaction et les taux de crossover et de mutation. Nous observerons également le nombre de générations nécessaires à l'apparition du réseau optimal en fonction du nombre de neurones du réseau et en fonction du nombre d'individus dans la population.

Poids des objectifs de la fitness

L'exemple du chapitre précédent sur la fonction *ou* a montré que la valeur donnée aux poids des trois objectifs de la fitness est importante. En effet, selon les coefficients donnés, le réseau final obtenu pour l'exécution n'est pas le même et peut ne pas être optimal pour les trois objectifs. Nous allons essayer de trouver les poids à donner aux trois critères de la fitness de façon à aboutir au réseau représentant un compromis acceptable. Pour nous, cela signifie qu'il doit parfaitement modéliser la fonction tout en possédant le moins de liens possibles et en prenant un minimum de temps.

Pour trouver les poids optimaux, nous allons exécuter notre algorithme génétique avec différentes combinaisons de poids. Nous lancerons 10 exécutions pour chaque combinaison de façon à obtenir un nombre moyen de générations nécessaires à l'apparition du réseau optimal. Nous analyserons ensuite les résultats obtenus et nous verrons quelles combinaisons sont à écarter. Pour éviter que cette analyse ne soit faussée par le caractère aléatoire de la population initiale, nous nous assurons que cette dernière est la même pour chaque exécution de l'algorithme. Dans le même but, nous utilisons notre algorithme génétique sans insérer de nouveaux individus à chaque génération.

Pour faciliter la référence aux poids de nos trois objectifs, nous allons les appeler :

- α_e = poids de la pénalité sur le nombre d'erreurs commises par rapport à la cible (objectif 1),
- α_{co} = poids de la pénalité sur le nombre de connexions (objectif 2),
- α_{tps} = poids de la pénalité sur le temps d'exécution (objectif 3).

Les graphes de la figure 5.2 représentent, pour une tâche donnée, les combinaisons de coefficients dont la moyenne de générations nécessaires à l'apparition du réseau optimal est inférieure au nombre maximal de générations en vert et les

autres en bleu. Cela signifie que les combinaisons de coefficients qui permettent d'arriver au réseau optimal avant le nombre maximal de générations pour au moins une exécution sont représentées en vert tandis que les autres sont représentées en bleu. Dans la figure de gauche, la tâche représentée est la tâche A tandis qu'à droite, il s'agit de la tâche B. Dans cette première analyse, nous nous intéressons aux poids α_e et α_{co}. Il n'est, en effet, pas nécessaire de représenter α_{tps}. Sa valeur va dépendre de celle des deux autres puisque la somme des coefficients doit valoir 1.

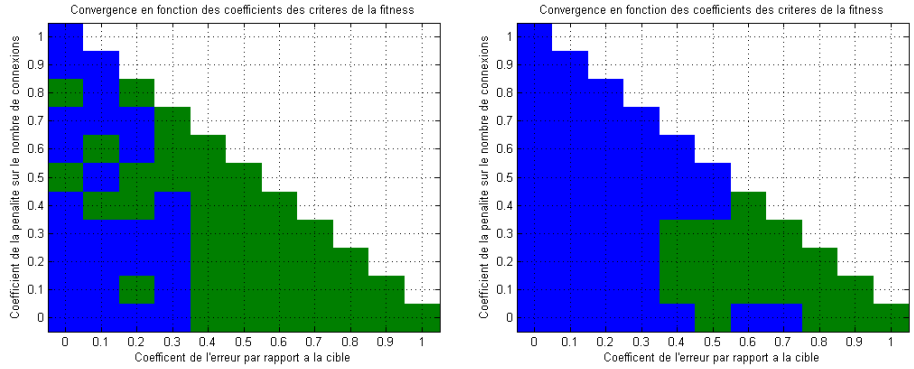

FIGURE 5.2 – Coefficients des objectifs et réseau optimal. Les combinaisons de coefficients qui permettent d'obtenir le réseau optimal avant la fin de l'algorithme pour au moins une exécution de celui-ci sont représentées en vert tandis que celles qui ne le permettent pas sont en bleu. Le graphe de gauche représente les résultats obtenus pour la tâche A et celui de droite ceux obtenus pour la tâche B.

Nous pouvons voir que l'importance accordée au poids de l'erreur commise par rapport à la cible (α_e) est grande. En effet, sa valeur doit être proche ou supérieure à 0.5 pour que le réseau optimal soit atteint. Cela signifie que le premier objectif détermine la moitié de la fitness de nos réseaux. Ce résultat est logique. En effet, notre but premier est de modéliser une tâche sans erreur. Il est donc normal que la moindre erreur soit sanctionnée de façon assez sévère par l'algorithme.

Si nous lançons notre algorithme sur les tâches C et D, nous obtenons les graphes de la figure 5.3. Ceux-ci ne nous permettent pas de réduire le nombre de combinaisons qui peuvent être utilisées pour obtenir le réseau optimal. Toutefois, nous pouvons voir de façon très claire sur ces deux graphes qu'il ne suffit pas d'avoir un poids important pour le premier objectif. En effet, si le poids de la pénalité sur le nombre de connexions est nul, c'est-à-dire si $\alpha_{co} = 0$, il est impossible de trouver le réseau optimal et ce quelle que soit la valeur de α_e.

Pour choisir la combinaison de poids que nous utiliserons par la suite, nous allons repérer les combinaisons qui permettent la convergence la plus rapide, c'est-à-dire l'apparition du réseau optimal après un nombre minimum de générations. Pour cela, nous commençons par recouper les analyses de nos quatre tâches pour obtenir les combinaisons qui permettent la convergence d'au moins une exécution pour chacune d'elles. Nous regardons ensuite le nombre minimum

de générations nécessaires à la convergence et nous retenons les combinaisons de poids qui se rapprochent de ce minimum parmi celles sélectionnées précédemment.

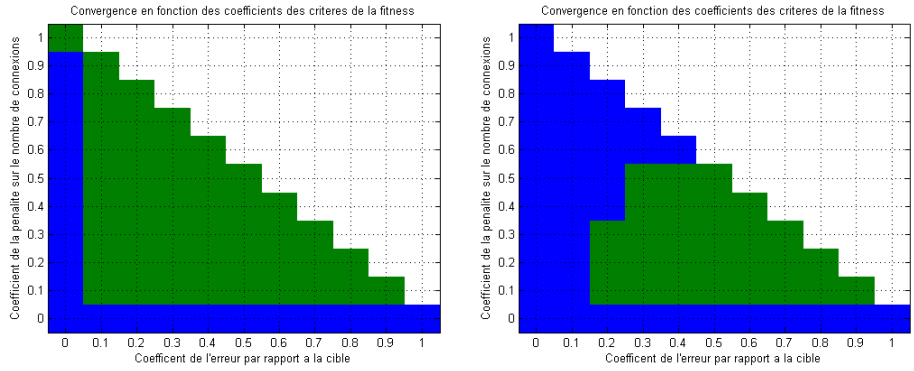

FIGURE 5.3 – Coefficients des critères et réseau optimal. Les combinaisons de coefficients qui permettent d'obtenir le réseau optimal avant la génération maximale pour au moins une exécution sont représentées en vert tandis que celles qui ne le permettent pas sont en bleu. Les tâches observées sont les tâches C et D.

Trois combinaisons de poids donnent des résultats de convergence assez rapide : $[0.5, 0.3, 0.2]$, $[0.6, 0.2, 0.2]$ et $[0.7, 0.2, 0.1]$. En effet, les résultats obtenus pour chacune de ces combinaisons sont repris dans le tableau 5.2.

	$[0.5, 0.3, 0.2]$	$[0.6, 0.2, 0.2]$	$[0.7, 0.2, 0.1]$
Tâche A	4.3	5.4	4
Tâche B	73.5	59.6	71.4
Tâche C	81.9	90.2	87.6
Tâche D	163.3	158.6	190.4

TABLE 5.2 – Nombres de générations nécessaires pour atteindre le réseau optimal en fonction de la tâche à réaliser et de la combinaison de poids utilisée.

Ces résultats sont assez similaires et nous pourrions utiliser chacune de ces combinaisons pour nos prochaines analyses. Nous choisissons de travailler avec la combinaison $[0.6, 0.2, 0.2]$ car celle-ci nous donne un poids assez important pour le premier objectif et ne favorise aucun des deux autres objectifs par rapport à l'autre.

En conclusion, lorsque nous passerons à l'analyse des réseaux optimaux, nous utiliserons la combinaison de poids suivante :

- $\alpha_e = 0.6$
- $\alpha_{co} = 0.2$
- $\alpha_{tps} = 0.2$

Interaction entre les objectifs

Nous avons vu dans l'exemple sur la fonction *ou* (voir chapitre précédent) que le fait d'ajouter la pénalité sur le temps d'exécution du réseau dans la fitness ne changeait pas le résultat. Dès lors, nous pouvons nous demander s'il n'existerait pas une corrélation entre nos trois objectifs qui rendrait l'ajout d'un de ceux-ci redondant par rapport aux deux autres.

Pour vérifier cette hypothèse, nous allons prendre une population aléatoire et nous allons calculer les valeurs des trois objectifs pour les individus qui en font partie. Nous tracerons ensuite des graphes des différents objectifs pour voir si nous pouvons trouver des corrélations entre ceux-ci.

Le graphe de la figure 5.4 est une représentation en trois dimensions de nos objectifs pour une population de réseaux de trois neurones qui doivent modéliser la fonction *ou*. Il semble y avoir un corrélation négative entre l'erreur commise par rapport à la cible et la pénalité sur le temps d'exécution.

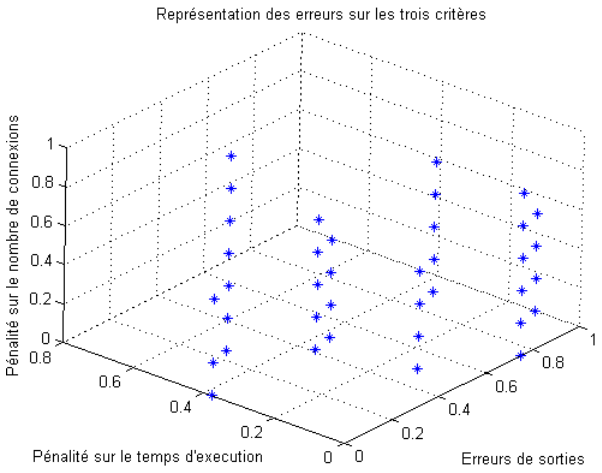

FIGURE 5.4 – Représentation des objectifs de la fitness pour une population de réseaux de neurones. La fonction à modéliser est la fonction *ou* et les réseaux sont composés de trois neurones. Il semble y avoir une corrélation négative entre l'erreur commise par rapport à la cible et la pénalité sur le temps d'exécution.

Pour quantifier cette corrélation, nous allons calculer les coefficients de corrélation ([16]) entre nos trois objectifs. La matrice obtenue est la suivante :

$$\begin{pmatrix} 1 & -0.40 & -0.85 \\ -0.40 & 1 & 0.47 \\ -0.85 & 0.47 & 1 \end{pmatrix}$$

Nous pouvons voir que le coefficient de corrélation entre l'erreur commise par rapport à la cible et la pénalité sur le temps d'exécution est de -0.85. Nous pouvons conclure qu'il existe bien une corrélation négative entre ces deux objectifs. Il est donc impossible de minimiser les deux en même temps, ce qui illustre la

notion de compromis introduite dans la description des algorithmes génétiques multi-objectifs.

Cette corrélation négative est assez simple à expliquer et découle de la façon dont nous avons initialisé notre algorithme. En effet, les états de tous les neurones qui ne sont pas des entrées sont initialisés à zéro. Or, si nous modélisons la fonction *ou*, trois combinaisons d'entrées sur quatre attendent une sortie de 1. Nous devons donc changer d'état pour 3 combinaisons. Comme la pénalité sur l'erreur est donnée par l'équation

$$1 - \frac{1}{4} \sum_{j=1}^{4} \frac{(trans_j + periode_j)}{2^{n-nb\_entrées}}$$

et que nos réseaux sont composés de trois neurones dont deux entrées, nous allons avoir une pénalité sur le temps d'exécution très forte pour les réseaux qui donnent les bonnes sorties. Notre objectif 3 aura donc une valeur assez petite (0.125).

La graphe de gauche de la figure 5.5 représente la relation entre nos deux objectifs et confirme ce que nous avions vu en trois dimensions. Le graphe de droite, quant à lui, représente ces mêmes objectifs mais pour une population qui cherche à modéliser la fonction *et*. Nous pouvons voir que la corrélation est cette fois positive. En effet, si nous calculons le coefficient de corrélation entre nos deux objectifs, nous obtenons une valeur de 0.85, ce qui confirme notre intuition. Encore une fois, cela nous semble logique puisque dans le cas de cette fonction, une seule combinaison d'entrées attend une sortie de 1 et seul un état doit donc changer. Nous aurons une valeur de 0.375 pour le troisième objectif si le réseau réalise correctement la tâche.

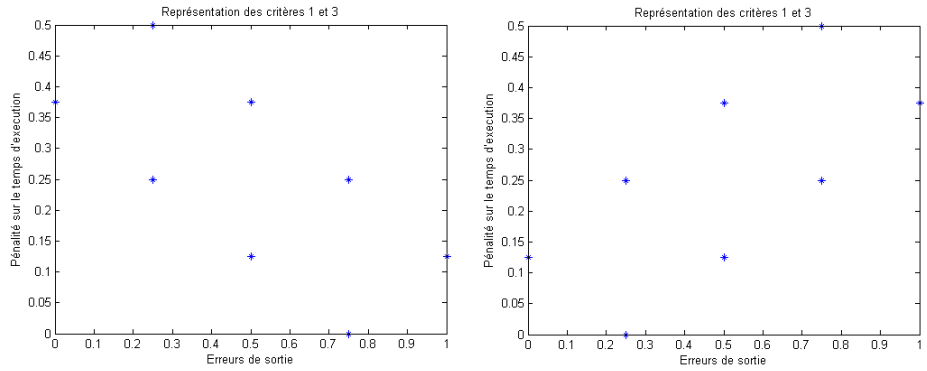

FIGURE 5.5 – Répartition des objectifs 1 et 3 de la fitness pour une population de réseaux de neurones. Les réseaux sont composés de trois neurones : deux entrées et une sortie. Ils ont pour objectif de modéliser la fonction *ou* à gauche et la fonction *et* à droite. Selon la fonction modélisée, la relation entre la pénalité sur le nombre d'erreurs commises et la pénalité sur le temps d'exécution change.

La différence de pénalité entre deux réseaux ayant une seule sortie différente est très marquée car ceux-ci sont très petits. Si nous prenons des réseaux pos-

sédant des neurones intermédiaires, le nombre d'états possibles est plus grand et la différence entre nos réseaux sera moins marquée. De ce fait, la corrélation entre la pénalité sur l'erreur commise par rapport à la cible et la pénalité sur le temps d'exécution sera plus petite pour des réseaux plus grands. Par exemple, si nous prenons une population de réseaux de six neurones qui ont pour but de modéliser la fonction *ou*, le coefficient de corrélation entre nos deux critères a une valeur de -0.31. La forte corrélation entre nos deux objectifs n'existe donc plus.

La relation que nous venons d'établir entre l'erreur commise par rapport à la cible et la pénalité sur le temps d'exécution est la seule visible dans la représentation en trois dimensions de la figure 5.4. Cette première intuition visuelle est confortée par la matrice de corrélation calculée précédemment. Cette dernière observation est tout à fait logique. En effet, il est possible d'avoir des réseaux de neurones qui réalisent une tâche sans erreur mais qui possèdent beaucoup de connexions inutiles. Il n'y a donc pas de liens entre ces deux objectifs. De la même façon, il n'existe pas de liens entre la pénalité sur le nombre de connexions et la pénalité sur le temps d'exécution. Cette absence de relation est également illustrée dans la figure 5.25. Le graphe de gauche représente la répartition des réseaux pour les objectifs 1 et 2. Celle de droite, quant à elle, représente la relation entre les objectifs 2 et 3. Une fois de plus, nous pouvons observer qu'il n'y a pas de corrélation entre ces objectifs.

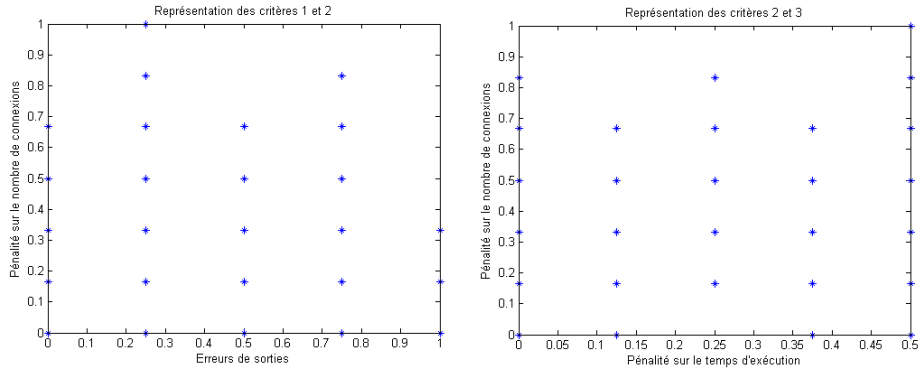

FIGURE 5.6 – Répartition des objectifs de la fitness pour une population de réseaux de neurones. Les réseaux sont composés de trois neurones et ont pour objectif de modéliser la fonction *ou*. La figure de gauche représente les objectifs 1 et 2 tandis que celle de droite représente les objectifs 2 et 3.

En conclusion, nous pouvons dire qu'il existe une corrélation entre l'erreur commise par rapport à la cible et la pénalité sur le temps d'exécution lorsque nous travaillons sur de petits réseaux de neurones. Cette corrélation dépend de la fonction qui doit être modélisée par ceux-ci. Lorsque la taille du réseau augmente, le nombre d'états possibles devient plus grand et la corrélation devient de moins en moins importante.

Taux de crossover et de mutation

Jusqu'à présent, nous avons exécuté notre algorithme avec des taux de crossover et de mutation fixés, à savoir 0.9 pour le taux de crossover et 0.1 pour celui de mutation. Nous allons essayer d'affiner ces deux valeurs pour optimiser notre algorithme.

Dans le but de trouver les taux de crossover et de mutation optimaux, nous allons exécuter notre algorithme en faisant varier ces deux paramètres et comparer les résultats obtenus. L'algorithme est exécuté 10 fois avec chaque combinaison de paramètres de façon à obtenir un nombre moyen de générations nécessaires à la convergence. Pour éviter que cette analyse ne soit faussée par le caractère aléatoire de la population initiale, nous nous assurons que cette dernière est la même pour chaque exécution de l'algorithme. Dans le même but, nous éliminons également l'ajout d'individus aléatoires.

Avant de passer à la recherche des paramètres optimaux, nous allons illustrer l'importance de nos deux paramètres dans la réussite de l'exécution. En effet, si nous traçons la moyenne de la fitness de la population finale en fonction de ceux-ci, et ce, quelle que soit la tâche réalisée, nous obtenons le même comportement que dans le graphe de la figure 5.7. Celui-ci représente les résultats obtenus pour l'exécution de la tâche B. Nous pouvons observer que la moyenne de la population finale semble augmenter lorsque le taux de mutation diminue (excepté pour zéro). Si nous calculons le coefficient de corrélation entre notre fitness moyenne et notre taux de mutation en enlevant les valeurs obtenues pour un taux de mutation nul, nous obtenons une corrélation de -0.97. Nous avons donc une forte corrélation négative entre la fitness moyenne de la population et le taux de mutation utilisé. Le coefficient de corrélation entre la fitness moyenne et le taux de crossover est quant à lui de 0.02. La fitness moyenne ne dépend donc pas du taux de crossover.

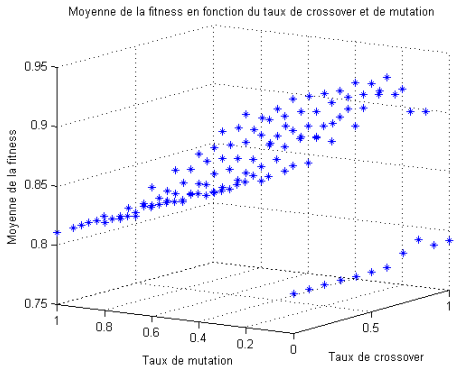

FIGURE 5.7 – Moyenne de la fitness pour la population finale. La tâche exécutée est la tâche B. Nous pouvons voir que la moyenne de la population augmente avec la diminution du taux de mutation tant que celui-ci n'est pas égal à zéro. Aucun comportement clair n'est observé en fonction du taux de crossover.

Nous pouvons donc conclure que le taux de mutation est le paramètre le plus important des deux puisqu'il influence fortement le comportement de la fitness. Cette observation est très intéressante. En effet, avant le développement des premiers algorithmes génétiques, il existait déjà des algorithmes basés sur l'évolution. Cependant, ceux-ci ne possédaient pas de crossover et le mécanisme d'évolution reposait uniquement sur la mutation. Par la suite, les scientifiques se sont étonnés que de tels algorithmes fonctionnent. Si nous nous référons à nos résultats, la mutation est bien le paramètre le plus important et il est tout à fait logique que ces algorithmes fonctionnent.

Passons à présent à l'analyse de nos différentes combinaisons de paramètres. Les graphes de la figure 5.8 représentent les combinaisons qui ont convergé vers le réseau optimal avant la génération finale pour au moins une exécution en vert et celles qui n'ont jamais atteint cet optimum avant la fin de l'algorithme en bleu. Le graphe de gauche représente les résultats obtenus pour l'exécution de la tâche C tandis que celui de droite représente ceux de la tâche D.

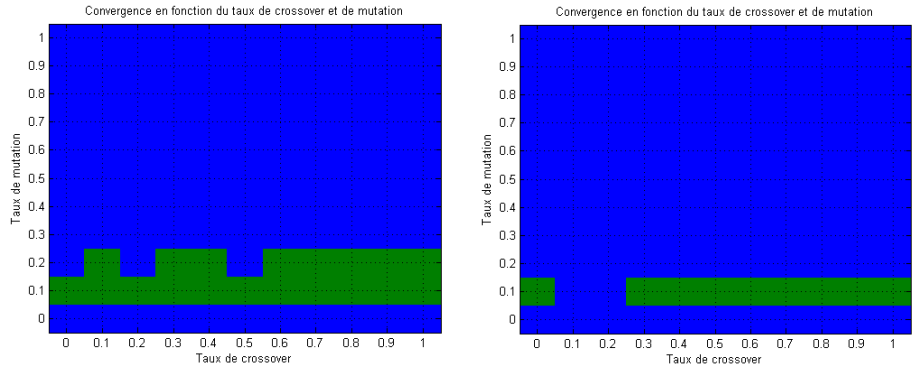

FIGURE 5.8 – Convergence vers le réseau optimal en fonction des taux de mutation et de crossover. Les combinaisons de paramètres pour lesquelles l'algorithme a atteint l'optimum avant la fin de l'algorithme pour au moins une exécution sont représentées en vert tandis que les autres sont représentées en bleu. La figure de gauche représente les résultats obtenus pour l'exécution de la tâche C. Celle de droite représente les résultats de la tâche D.

Nous pouvons voir que le taux de mutation qui permet la convergence dans les deux exécutions est 0.1 tandis le taux de crossover peut varier entre 0.3 et 1. Toutefois, le taux de mutation est généralement plus petit dans la littérature sur les algorithmes génétiques. Nous allons donc relancer notre programme en le ciblant entre 0 et 0.1 par pas de 0.01 pour le taux de mutation et entre 0.3 et 1 pour le taux de crossover.

Le graphe de la figure 5.9 représente les résultats de l'exécution de la tâche D pour les combinaisons de paramètres précités. Nous avons choisi de présenter les résultats de cette tâche car les paramètres qui permettaient de faire converger les réseaux dans la figure 5.8 étaient moins nombreux que pour la tâche C, ce qui signifie que l'optimisation est plus difficile à réaliser.

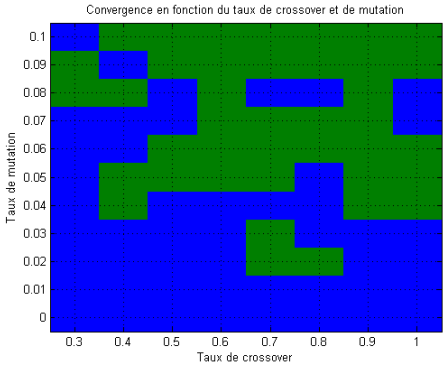

FIGURE 5.9 – Convergence vers le réseau optimal en fonction du taux de crossover et de mutation. Les résultats présentés sont ceux obtenus pour l'exécution de la tâche D. Les combinaisons pour lesquelles l'algorithme génétique trouve le réseau optimal sont représentées en vert.

Nous pouvons voir qu'un certain nombre de combinaisons permette la convergence en un nombre limité de générations. Pour la plupart d'entre-elles, le taux de mutation est supérieur à 0.05. Pour choisir celle qui nous convient, nous regardons plus précisément à quelle génération chacune de ces combinaisons converge en moyenne. La combinaison de paramètres qui demande le moins de générations pour converger est la suivante :

- taux de crossover : 0.9
- taux de mutation : 0.09

Cette combinaison donne des résultats tout à fait satisfaisants pour les exécutions lancées sur les autres fonctions. En conclusion, nous garderons cette combinaison de paramètres pour la suite de nos analyses.

Taille du réseau et nombre de générations

Passons à présent à l'analyse de la relation qui existe entre la taille du réseau et le nombre de générations nécessaires à sa convergence. En effet, il est évident que les réseaux de petite taille convergent rapidement vers un réseau optimal puisqu'ils ont peu de combinaisons de connexions possibles tandis que les réseaux de grande taille mettront plus de temps à explorer toutes les combinaisons possibles et donc plus de temps à converger. De nouveau, le mot convergence aura le sens d'apparition du réseau optimal.

Pour réaliser cette analyse, nous modélisons certaines fonctions pour lesquelles nous connaissons les réseaux optimaux à l'aide de réseaux de taille croissante. Nous arrêtons chaque exécution de l'algorithme génétique dès l'apparition de l'individu optimal. Le nombre maximal de générations est 5000. Pour chaque taille de réseau, nous exécutons l'algorithme dix fois de façon à obtenir un nombre moyen de générations nécessaires à la convergence. Contrairement aux analyses précédentes, nous ne supprimons pas l'ajout des individus aléatoires à chaque génération puisque cela ne peut pas biaiser notre analyse.

Le graphe de gauche de la figure 5.10 illustre les résultats obtenus pour des réseaux qui ont pour objectif de modéliser la fonction *ou*. Celui de droite, quant à lui, fait de même pour des réseaux à deux sorties qui doivent respectivement modéliser les fonctions *ou* et *et*.

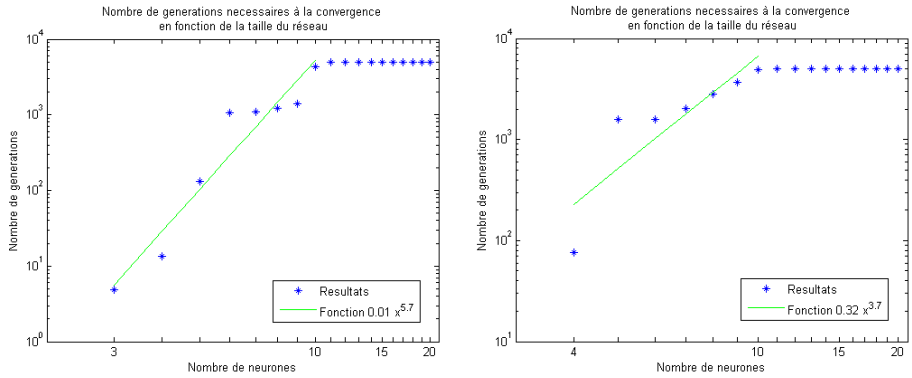

FIGURE 5.10 – Nombre de générations nécessaires à la convergence en fonction de la taille du réseau. La figure de gauche représente les résultats obtenus pour des réseaux qui ont pour but de modéliser la fonction *ou*. A droite, les réseaux ont deux sorties qui doivent respectivement modéliser les fonctions *ou* et *et*. La limite de 5000 générations permet la convergence des réseaux de maximum 10 neurones. Dans les deux graphes, les résultats suivent une fonction polynomiale.

Le comportement global observé est le même pour les deux fonctions. En effet, il est possible de trouver la solution de notre modélisation pour des réseaux contenant au maximum 10 neurones. Les réseaux de taille supérieure atteignent la limite des 5000 générations sans avoir trouvé le réseau optimal modélisant la fonction. Nous pouvons cependant noter que la fonction logique simple *ou* semble plus facile à modéliser que les fonctions *ou* et *et* ensembles, vu la différence de générations observées pour chacune des tailles de réseau.

Si nous extrapolons une courbe à partir des résultats obtenus, nous obtenons que le nombre de générations nécessaires à la convergence du réseau de neurones croit de façon polynomiale avec la taille de ce réseau. Nous pouvons donc conclure que le nombre de générations nécessaires à la convergence de réseaux de taille supérieure à 10 sera assez long. Nous nous limiterons donc à des réseaux plus petits dans cette partie du travail, ainsi que dans l'analyse qui suivra sur les réseaux optimaux eux-mêmes.

Taille de la population et nombre de générations

Lors des exécutions précédentes de notre algorithme, nous avons travaillé avec une population de 100 individus. Nous aimerions maintenant savoir s'il existe une corrélation entre le nombre de générations nécessaires à l'apparition du meilleur individu et la taille de notre population. En effet, il nous semble évident qu'une population de 10 individus aura plus de mal à converger qu'une population de taille 100 puisque les réseaux sont moins nombreux et permettent moins de possibilités de combinaisons de connexions. Est-il pour autant utile

de prendre une grande population dans l'espoir de converger plus rapidement ? N'atteint-on pas un plafond à partir d'une certaine taille de population ? Il faut également penser au temps d'exécution qui augmente en fonction du nombre d'individus utilisés. Nous allons donc tenter de trouver un compromis entre rapidité de convergence et rapidité d'exécution de chaque itération.

Pour répondre à ces questions, nous exécutons l'algorithme de façon à obtenir un réseau optimal qui modélise une fonction donnée. Nous effectuons cette exécution pour des populations de taille croissante et nous comparons ensuite les résultats obtenus. Pour minimiser l'intervention aléatoire due à l'initialisation de la population dans notre analyse, nous lançons dix exécutions pour chacune des tailles de la population de façon à obtenir un résultat moyen de convergence.

Le graphe de gauche de la figure 5.11 représente le nombre moyen de générations nécessaires à l'apparition du meilleur individu pour l'exécution de la tâche A. A droite, la tâche réalisée est la tâche C.

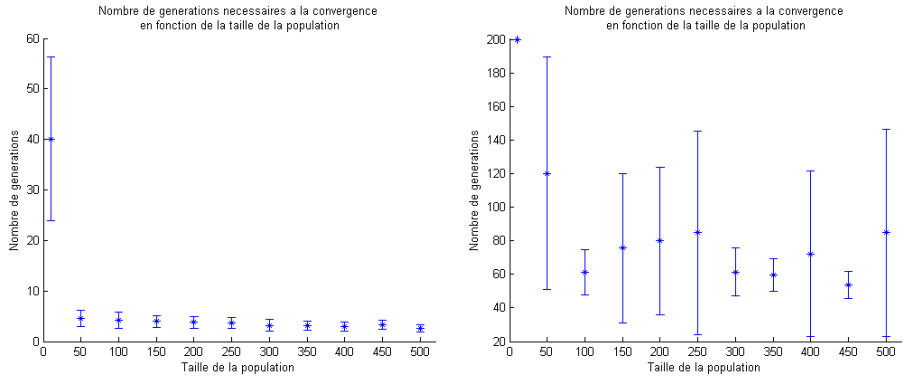

FIGURE 5.11 – Nombre de générations nécessaires à la convergence en fonction du nombre d'individus de la population. Les réseaux doivent modéliser la fonction *ou*. La figure de gauche représente les résultats obtenus pour des réseaux de trois neurones. A droite, les réseaux sont composés de six neurones.

Sur le graphe de gauche, nous pouvons voir que le réseau optimal est très rapidement atteint, et ce pour une population de 50 individus ou plus. Cela nous semble logique. En effet, les réseaux étudiés sont très petits et toutes les topologies possibles peuvent être étudiées très rapidement. Sur celle de droite, nous pouvons voir que le nombre moyen de générations nécessaires à la convergence diminue jusqu'à atteindre une population de 100 individus. Après cela, il se stabilise. Il semblerait donc qu'il soit inutile de prendre une population trop grande lorsque nous utilisons notre algorithme génétique. Nous pouvons remarquer grâce aux bornes d'erreur que le nombre de générations nécessaires à la convergence peut varier assez fortement d'une exécution à l'autre. Cette variation est due au caractère stochastique des algorithmes génétiques.

Pour confirmer notre intuition, nous allons de nouveau lancer cette analyse pour des réseaux de dix neurones qui doivent modéliser la fonction *ou*. Le graphe ob-

tenu est présenté dans la figure 5.12. Nous pouvons voir que le nombre moyen de générations nécessaires à la convergence décroît jusqu'à atteindre une population de 200 individus. Ensuite, il n'y a plus d'améliorations visibles. Nous pouvons donc affirmer qu'il est inutile de prendre une population de taille trop importante lors de l'exécution de notre algorithme génétique.

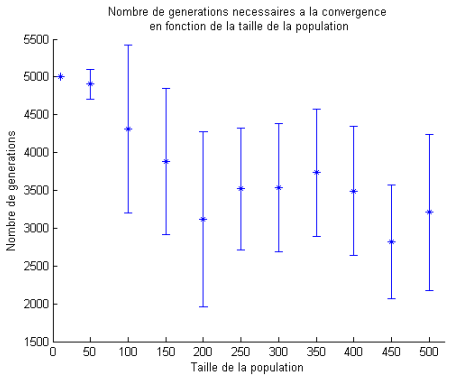

FIGURE 5.12 – Nombre de générations nécessaires à la convergence en fonction du nombre d'individus de la population (2). Les réseaux sont composés de dix neurones et doivent modéliser la fonction *ou*.

En conclusion, il n'est pas envisageable au vu des résultats dont nous disposons d'augmenter fortement la taille de la population pour avoir une convergence plus rapide. Lorsque nous passerons à la phase suivante d'analyse, nous prendrons une population de 200 individus, ce qui nous permettra d'avoir une convergence assez rapide et ce sans trop ralentir l'exécution de chaque itération de notre algorithme.

5.1.2 Algorithme génétique multi-objectif NSGA

Passons à présent à l'analyse de notre algorithme génétique NSGA. Dans celui-ci, nous ne devons plus donner de poids à nos trois objectifs. Les paramètres à analyser seront donc les taux de crossover et de mutation. Comme précédemment, nous observerons aussi le nombre de générations nécessaires à l'apparition du réseau optimal en fonction de la taille du réseau et de la taille de la population utilisée.

Taux de crossover et de mutation

Comme pour l'algorithme précédent, nous avons testé le fonctionnement de notre algorithme avec un taux de crossover de 0.9 et un taux de mutation de 0.1. Cette analyse va nous permettre d'affiner ces valeurs.

La procédure suivie pour obtenir nos paramètres optimaux est la même que pour l'algorithme génétique à fitness pondérée. En effet, l'algorithme est exécuté trois fois pour chaque combinaison de paramètres de façon à obtenir un nombre moyen de générations nécessaires à l'apparition du réseau optimal. Nous

comparons ensuite les résultats obtenus de façon à choisir la meilleure combinaison possible de paramètres. La population initiale de chaque exécution est la même de façon à ne pas fausser notre analyse. De même, l'ajout d'individus aléatoires à chaque génération est supprimé.

Les graphes de la figure 5.13 représentent les résultats obtenus pour chaque combinaison de paramètres. Celles qui ont permis la convergence vers le réseau optimal avant le nombre maximal de générations pour au moins une exécution sont représentées en vert tandis que les autres sont représentées en bleu. Le graphe de gauche représente les résultats obtenus pour l'exécution de la tâche C. Celui de droite, quant à lui, représente les résultats de la tâche D.

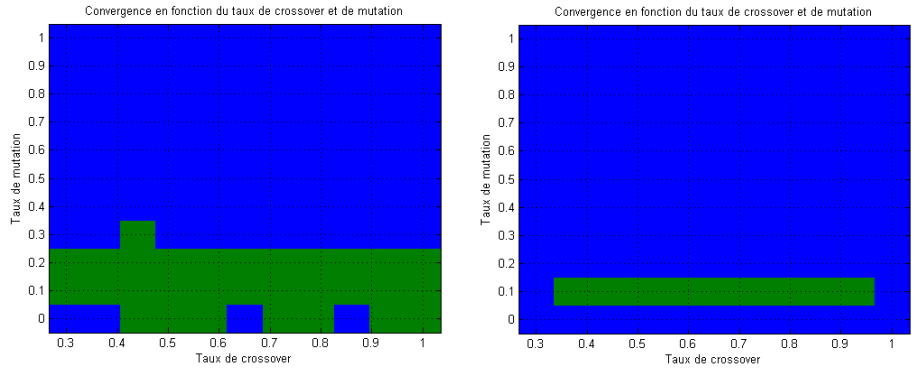

FIGURE 5.13 – Convergence vers le réseau optimal en fonction des taux de mutation et de crossover. Les combinaisons de paramètres pour lesquelles l'algorithme a atteint l'optimum avant la génération finale pour au moins un exécution de l'algorithme sont représentées en vert tandis que les autres sont représentées en bleu. La figure de gauche représente les résultats obtenus pour l'exécution de la tâche C. Celle de droite représente ceux de la tâche D.

Nous observons que le taux de mutation qui permet la convergence pour les deux types de réseaux analysés est 0.1. Le taux de crossover, quant à lui, varie entre 0.1 et 0.9. Nous pouvons en déduire que la valeur du taux de mutation doit être assez précise tandis que celle du taux de crossover n'est pas très importante pour la convergence. Comme pour l'algorithme précédent, nous allons relancer notre programme en ciblant les combinaisons pour lesquelles le taux de mutation est compris entre 0 et 0.1 par pas de 0.01 et pour lesquelles le taux de crossover est compris entre 0.1 et 0.9. Nous verrons ainsi si nous pouvons affiner les valeurs de notre combinaison de paramètres pour rendre notre algorithme le plus efficace possible.

Les résultats obtenus pour l'exécution de la tâche D sont représentés dans la figure 5.14. De nouveau, les combinaisons permettant la convergence vers le réseau optimal avant la génération finale pour au moins une exécution de l'algorithme sont représentées en vert tandis que les autres sont en bleu.

Nous pouvons voir qu'il existe plusieurs combinaisons du taux de crossover et de mutation qui permettent la convergence. Notons que dans la majorité d'entre-

elles, le taux de mutation est supérieur à 0.05. Nous allons à présent regarder le nombre moyen de générations nécessaires à la convergence pour chacune de ces combinaisons. De cette façon, nous pourrons choisir celle qui nous convient le mieux. La combinaison qui converge le plus rapidement est celle dont la valeur du taux de mutation est 0.09 et dont la valeur du taux de crossover est de 0.8. Nous utiliserons donc cette combinaison pour la suite de nos analyses.

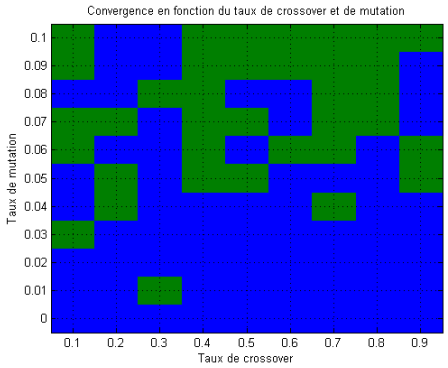

FIGURE 5.14 – Convergence vers le réseau optimal en fonction du taux de crossover et de mutation. La tâche exéctutée est la tâche D. Les combinaisons pour lesquelles l'algorithme génétique trouve le réseau optimal avant la génération finale pour au moins une exécution de l'algorithme sont représentées en vert.

Taille du réseau et nombre de générations

Analysons à présent la relation entre la taille du réseau et le nombre de générations nécessaires à sa convergence. Pour réaliser cette analyse, nous allons utiliser la même procédure que pour l'algorithme génétique à fitness pondérée. Nous allons faire modéliser certaines fonctions par des réseaux de taille croissante et nous allons regarder le nombre moyen de générations nécessaires à la convergence pour chacune de ces tailles. Le nombre maximal de générations est 5000. Contrairement à l'analyse sur les taux de crossover et de mutation, nous ne supprimons pas l'ajout d'individus aléatoires à chaque génération.

Les résultats obtenus sont représentés dans la figure 5.15. Le graphe de gauche illustre les résultats obtenus pour des réseaux qui ont pour objectif de modéliser la fonction *ou*. Celui de droite, quant à lui, fait de même pour des réseaux à deux sorties qui doivent modéliser les fonctions *ou* et *et*. Comme l'algorithme génétique à fitness pondérée ne nous a pas permis de converger pour des réseaux de taille supérieure à 10 neurones, nous commençons cette analyse pour des réseaux de cette taille au maximum. Nous élargirons notre analyse par la suite si nécessaire.

Nous pouvons observer le même comportement pour nos deux fonctions. En effet, la limite de 5000 générations permet de trouver la solution optimale de notre modélisation pour des réseaux de maximum 10 neurones. Notons que les réseaux qui doivent modéliser la fonction *ou* ont une convergence un peu plus rapide que ceux qui doivent modéliser les fonctions *ou* et *et* avec le même nombre

de neurones. Cela nous semble logique. En effet, la fonction *ou* est très simple et le réseau ne possède qu'une sortie et deux liens tandis que pour les fonctions *ou* et *et*, le réseau a besoin de deux sorties et de quatre liens. Cette seconde modélisation est donc un peu plus difficile à atteindre que la première. Nous pouvons également remarquer que le nombre moyen de générations nécessaires à la convergence des réseaux augmente de façon exponentielle avec la taille de ceux-ci.

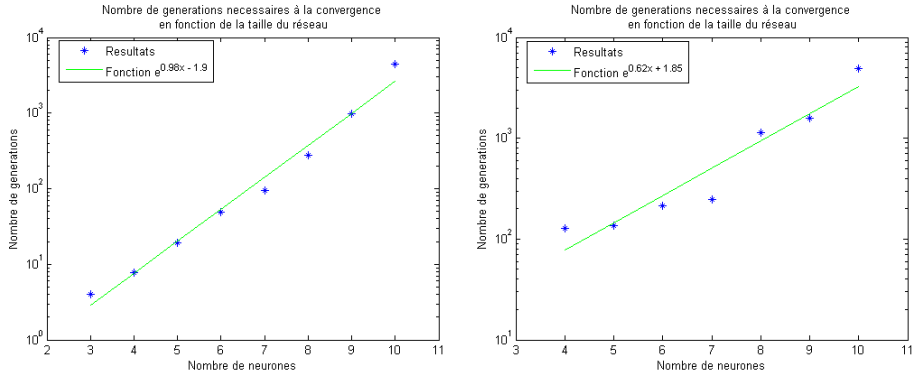

FIGURE 5.15 – Nombre de générations nécessaires à la convergence en fonction de la taille du réseau. La figure de gauche représente les résultats obtenus pour des réseaux qui ont pour tâche de modéliser la fonction *ou*. A droite, les réseaux ont deux sorties qui doivent modéliser les fonctions *ou* et *et*. Le nombre moyen de générations nécessaires à la convergence est une fonction exponentielle du nombre de neurones du réseau.

En conclusion, le nombre de générations nécessaires à la convergence de réseaux de plus de dix neurones est assez grand. Il faudra donc un certain temps avant d'atteindre le réseau optimal.

Taille de la population et nombre de générations

Jusqu'à présent, nous avons lancé nos exécutions avec une population de 100 individus. Nous allons maintenant voir s'il est plus intéressant de travailler avec une population de taille différente. Notons toutefois que le temps d'exécution augmente avec l'augmentation de la taille de la population puisque notre algorithme doit trier les individus en fonction de leur degré de non-dominance avant de leur attribuer une fitness. Nous allons donc, comme pour l'algorithme précédent, tenter de trouver un compromis entre rapidité de convergence et rapidité d'exécution de chaque itération.

Pour trouver ce compromis, nous exécutons l'algorithme génétique de façon à obtenir un réseau optimal qui modélise une fonction donnée, et ce pour des populations de taille croissante. Trois exécutions sont lancées pour chacune des tailles dans le but d'obtenir un résultat moyen de convergence.

Les graphes de la figure 5.16 représentent les résultats obtenus pour l'exécution de la tâche A à gauche et de la tâche C à droite.

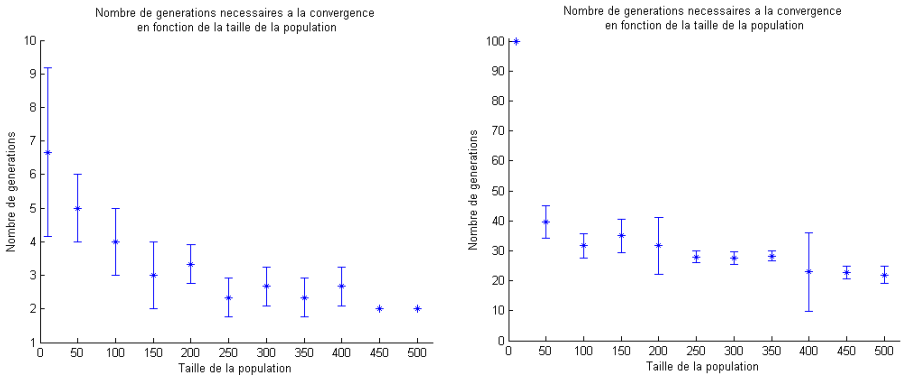

FIGURE 5.16 – Nombre de générations nécessaires à la convergence en fonction du nombre d'individus de la population. La figure de gauche représente les résultats obtenus pour l'exécution de la tâche A. A droite, la tâche réalisée est la tâche C.

Si nous analysons la figure de gauche, nous pouvons observer que le réseau optimal est très rapidement atteint et ce quelle que soit la taille de la population. Ce résultat est dû à la simplicité de la fonction et à la petite taille du réseau. En ce qui concerne la figure de droite, nous pouvons voir que la convergence est possible avec une population de minimum 50 individus. Le nombre de générations nécessaires à la convergence décroît légèrement lorsque la taille de la population augmente. Cette décroissance n'est toutefois pas suffisante pour nous inciter à prendre une population plus importante pour la suite de nos analyses. En effet, le temps nécessaire au tri des solutions non-dominées et au calcul de la fitness de chaque individu augmente lorsque la taille de la population augmente (voir figure 5.17). Cette augmentation suit une courbe polynomiale. Il n'est donc pas utile de prendre une population plus grande car le temps gagné sur le nombre de générations nécessaires à la convergence est perdu lors du tri des individus.

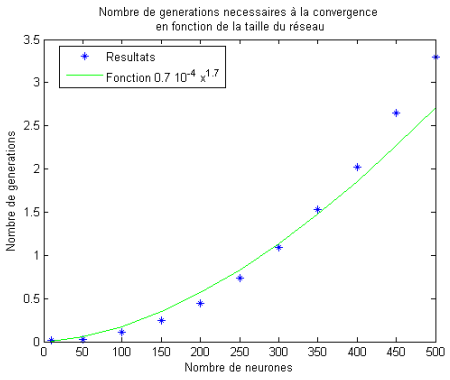

FIGURE 5.17 – Temps nécessaire au tri et au calcul de la fitness des individus en fonction de leur nombre. Plus la population est grande, plus il faut de temps pour réaliser cette étape importante de l'algorithme.

En conclusion, nous allons garder une population de 100 individus dans la suite de nos analyses. Cette taille de population nous donne une convergence satisfaisante et le tri des solutions non-dominées ne demande pas trop de temps.

5.1.3 Comparaison entre nos deux algorithmes

Nous allons à présent comparer brièvement nos algorithmes. Nous commencerons par analyser la distribution des valeurs prises par les trois objectifs pour la population finale de ces deux algorithmes. Nous verrons ensuite si les paramètres optimaux pour ceux-ci sont les mêmes. Nous comparerons également le nombre de générations nécessaires à leur convergence en fonction de la taille des réseaux de neurones. Nous terminerons en comparant leur taux de performance à chaque génération.

Commençons par observer la distribution des valeurs des trois objectifs pour les individus de la population finale de nos deux algoithmes. Cela nous permettra d'insister une fois de plus sur la principale différence qui existe entre ceux-ci. Les résultats présentés dans la figure 5.18 sont ceux obtenus pour la réalisation de la tâche A. Le graphe de gauche illustre les résultats de l'algorithme génétique à fitness pondérée et celui de droite les résultats de l'algorithme génétique multi-objectif NSGA. Chaque barre correspond à une seule valeur de l'objectif.

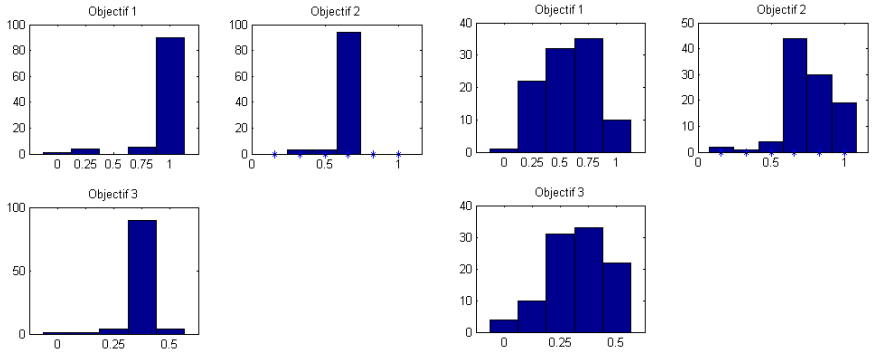

FIGURE 5.18 – Distribution des valeurs des trois objectifs pour la population finale de nos deux algorithmes. Chaque barre correspond à une seule valeur prise par les objectifs. Nous pouvons voir que l'algorithme génétique à fitness pondérée converge vers une seule solution, ce qui n'est pas le cas pour l'algorithme génétique multi-objectif NSGA.

Nous pouvons voir dans la figure de gauche que plus de 90% des individus ont la même valeur pour les trois objectifs. Cela signifie que l'algorithme génétique à fitness pondérée converge vers une seule solution optimale en fonction des poids assignés aux différents objectifs. Les individus qui ne possèdent pas les mêmes valeurs pour les trois objectifs sont ceux qui ont été insérés à la fin de l'itération. Dans la figure de droite, nous pouvons voir que la distribution des valeurs est plus étalée. Cette observation est logique puisque l'algorithme génétique multi-objectif NSGA ne converge pas vers une solution mais vers plusieurs. L'analyse de ce graphe nous a donc permis de rappeler cette différence très importante

qui existe entre les deux algorithmes génétiques étudiés.

Passons à la comparaison des paramètres communs à nos deux algorithmes. Ceux-ci sont repris dans le tableau 5.3.

	Fitness pondérée	NSGA
Taux de mutation	0.09	0.09
Taux de crossover	0.9	0.8
Taille de la population	200	100

TABLE 5.3 – Valeurs optimales des paramètres des deux algorithmes génétiques.

Nous pouvons voir que le taux de mutation est le même pour nos deux algorithmes. Le taux de crossover, quant à lui, est légèrement différent. Toutefois, nous avons vu au cours de nos analyses que ce paramètre n'est pas très important. En ce qui concerne la taille de la population, nous avons le double d'individus dans l'algorithme à fitness pondérée. Nous aurions également pu choisir une population de 200 individus pour l'algorithme génétique NSGA mais cela entraînait une forte augmentation du temps nécessaire au tri des individus en fonction de leur non-dominance. En conclusion, nous pouvons dire que, malgré quelques légères différences, les paramètres optimaux de nos deux algorithmes sont assez similaires.

Regardons à présent le nombre de générations nécessaires à la convergence de nos deux algorithmes sur la même optimisation, à savoir la modélisation des fonctions *ou* et *et* par nos réseaux. Les résultats obtenus sont représentés dans la figure 5.19.

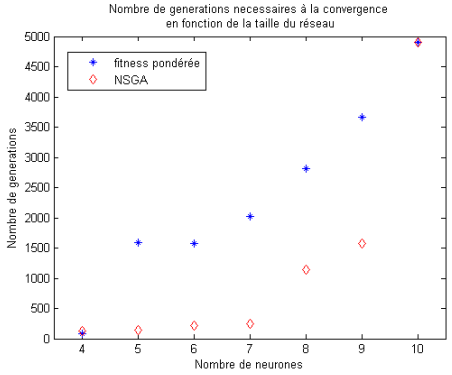

FIGURE 5.19 – Comparaison du nombre de générations nécessaires à la convergence pour nos deux algorithmes. Les résultats obtenus sont ceux de réseaux cherchant à modéliser les fonctions *ou* et *et*. Nous pouvons observer que l'algorithme NSGA converge plus rapidement pour la plupart des tailles du réseau.

Nous pouvons voir que l'algorithme génétique NSGA converge plus rapidement pour la plupart des tailles de réseau. Toutefois, le nombre de générations nécessaires à la convergence pour des réseaux de 10 neurones est équivalent pour

les deux algorithmes. Nous avons vu que la croissance du nombre de générations nécessaires à la convergence des réseaux suit une courbe polynomiale pour l'algorithme génétique à fitness pondérée et une courbe exponentielle pour l'algorithme NSGA. La croissance sera donc plus rapide pour ce second algorithme.

Nous venons de voir que l'algorithme génétique NSGA est plus rapide au niveau du nombre de générations nécessaires à la convergence pour la plupart des tailles de réseaux. Cependant, cela ne signifie pas que cet algorithme soit plus rapide au niveau du temps d'exécution. En effet, une itération, c'est-à-dire une génération, de l'algorithme génétique à fitness pondérée est exécutée en 0.15 secondes tandis pour l'agorithme NSGA, le temps nécessaire est de 5.75 secondes. Cette différence est due au tri des individus en fonction de leur non-dominance et au calcul de la fitness. L'algorithme génétique à fitness pondérée sera donc généralement plus rapide au niveau du temps d'exécution même s'il demande plus de générations pour converger.

Terminons cette comparaison en regardant le taux de performance de nos deux algorithmes. Pour cela, nous allons lancer dix exécutions d'une tâche donnée. Nous regarderons ensuite le nombre d'exécutions pour lesquelles la solution optimale a été découverte après x générations. Nous comparerons ensuite les résultats obtenus pour nos deux algorithmes.

Le graphe de gauche de la figure 5.20 représente les résultats obtenus pour l'exécution de la tâche A. Le graphe de droite, quant à lui, donne les résultats pour l'exécution de la tâche C.

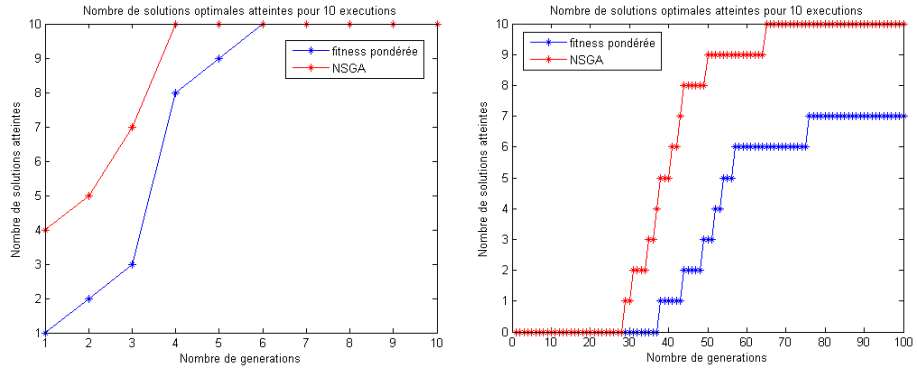

FIGURE 5.20 – Taux de performance de nos algorithmes. Les résultats représentés sont ceux obtenus lors de l'exécution de la tâche A à gauche et C à droite.

Dans le graphe de gauche, nous pouvons observer que nos deux algorithmes permettent la convergence des dix exécutions de l'optimisation. L'algorithme NSGA est toutefois plus efficace au niveau du nombre de générations nécessaires à la convergence. En effet, la solution optimale est trouvée au plus tard à la quatrième génération tandis que pour l'algorithme génétique avec poids, il faut attendre la sixième génération. De plus, si nous regardons la première génération (différente de la génération initiale), nous pouvons voir que le réseau

optimal est trouvé dans quatre exécutions sur dix pour l'algorithme NSGA. Pour l'algorithme à fitness pondérée, une seule exécution est optimisée à cette génération. Cela confirme que l'algorithme génétique multi-objectif NSGA est plus rapide que l'algorithme à fitness pondérée au niveau du nombre de générations nécessaires à sa convergence.

Nous pouvons voir dans le graphe de droite que les résultats obtenus pour nos deux algorithmes génétiques sont assez différents. L'algorithme génétique à fitness pondérée ne permet pas la convergence de toutes nos exécutions vers le réseau optimal après cent générations. En effet, seules sept d'entre-elles ont convergé. Au contraire, toutes les exécutions ont convergé après 70 générations pour l'algorithme génétique multi-objectif NSGA.

En conclusion, nous pouvons affirmer grâce à l'analyse du nombre de générations nécessaires à la convergence de nos réseaux et l'analyse du taux de performance que l'algorithme génétique multi-objectif NSGA demande moins de générations que l'algorithme génétique à fitness pondérée pour converger. Il n'est cependant pas plus rapide au niveau du temps d'exécution puisqu'une itération est beaucoup plus lente pour ce second algorithme. Nous continuerons à utiliser nos deux algorithmes pour la phase suivante d'analyse. Cela nous permettra de comparer les résultats obtenus et de vérifier si nous obtenons le même réseau optimal pour une optimisation donnée.

5.2 Analyse des réseaux optimaux

Après avoir défini les paramètres de nos algorithmes génétiques de façon plus précise, nous allons pouvoir les utiliser pour analyser nos réseaux. Notre objectif est de répondre à quelques questions. Tout d'abord, nous avons vu dans le deuxième chapitre qu'il était possible de modéliser toutes les fonctions logiques avec un perceptron. Or le modèle de réseaux de neurones implémenté est une simplification de ce perceptron. Est-il possible de modéliser toutes les fonctions logiques avec notre implémentation ?

Ensuite, imaginons que nous devons réaliser les tâches *et* et *ou* dans un réseau à quatre entrées et deux sorties. Est-il plus avantageux de prendre le meilleur réseau modélisant la fonction *et*, faire de même avec la fonction *ou* et de les assembler ou bien de chercher le meilleur réseau qui réalise les deux tâches en même temps ? La réponse à cette question nous permettra de réfléchir à une question importante, à savoir, existe-t-il une zone du cerveau indépendante pour chaque tâche ou certaines tâches sont-elles réalisées par une même zone ?

Nous profiterons de ces différentes analyses pour étudier la dégénérescence et la redondance de nos réseaux de neurones [28]. La dégénérescence est définie comme la capacité qu'ont des éléments structurellement différents de modéliser la même fonction. La redondance, quant à elle, survient quand la fonction est toujours modélisée par le même élément. Nous verrons s'il est possible de trouver des réseaux différents capables de réaliser la même tâche ou si nous tombons toujours sur la même fonction.

5.2.1 Implémentation des fonctions logiques

Pour voir si nous pouvons modéliser toutes les fonctions logiques à l'aide de nos réseaux, nous allons partir des fonctions logiques à une et deux entrées. Nous verrons si nous pouvons en déduire un comportement pour les fonctions logiques ayant un nombre d'arguments plus important. Notre but est de trouver, pour chaque fonction analysée, le nombre minimal de neurones nécessaires à sa modélisation. Nous utiliserons évidemment les trois objectifs définis précédemment de façon à obtenir un réseau optimal au point de vue du nombre de connexions et du temps d'exécution.

Fonctions logiques à une entrée

Il existe quatre fonctions logiques à une entrée. Celles-ci sont représentées dans le tableau 5.4.

x_1	f_0	f_1	f_2	f_3
0	0	0	1	1
1	0	1	0	1

TABLE 5.4 – Fonctions logiques à une entrée.

La modélisation de la fonction f_0, c'est-à-dire la fonction *nulle*, ne nécessite la présence d'aucun lien entre l'entrée et la sortie. Les trois autres fonctions, quant à elles, sont modélisées à l'aide d'une seule connexion partant de l'entrée vers la sortie. Le poids de cette connexion est 1 pour les fonctions f_1 et f_3 et -1 pour la fonction f_2. Aucune de ces fonctions ne nécessite de neurones intermédiaires pour être modélisée. Les deux types d'architectures obtenues lors de nos exécutions sont présentés dans la figure 5.21.

FIGURE 5.21 – Architectures des réseaux modélisant les fonctions logiques à une entrée. Le réseau de gauche est un réseau sans connexion qui permet de modéliser la fonction nulle. Les trois autres fonctions sont de la forme du réseau de droite. La fonction représentée est la négation (fonction f_2).

Fonctions logiques à deux entrées

Nous avons déjà présenté un tableau contenant toutes les fonctions logiques à deux entrées lors de l'analyse des neurones construits sur le modèle du perceptron (voir tableau 2.6). Ces fonctions sont au nombre de 16. Plusieurs types d'architectures peuvent être observés lorsque nous les modélisons à l'aide de nos réseaux de neurones. Nous allons donc présenter une architecture à la fois et expliquer brièvement d'où elle vient.

Comme pour les fonctions logiques à une entrée, la fonction *nulle* à deux entrées f_0 ne demande la présence d'aucune connexion pour être modélisée. En effet,

puisque le réseau ne possède pas de liens entre ses entrées et sa sortie et que tous les neurones autres que les entrées sont initialisés à 0, il est impossible que le neurone de sortie soit excité et qu'il ait une valeur de 1. De plus, ce réseau est bien celui qui minimise le nombre de connexions et le temps d'exécution. L'architecture du réseau modélisant cette fonction est représentée dans la figure 5.22. Nous pouvons observer que notre réseau est simplement composé de neurones isolés.

FIGURE 5.22 – Architecture des réseaux modélisant la fonction *nulle* à deux entrées. Il n'y a aucune connexion entre les entrées et la sortie. De cette façon, la sortie ne peut jamais avoir une valeur de 1 (il n'est jamais excité). Le réseau est composé de neurones isolés.

Nous observons ensuite une série de fonctions pour lesquelles le réseau est composé uniquement des deux entrées et de la sortie et pour lesquelles seule une des deux entrées est reliée à la sortie. Nous obtenons une telle modélisation pour les différentes fonctions présentées dans le tableau 5.5.

x_1	x_2	f_1	f_2	f_3	f_4	f_5
0	0	0	1	0	1	1
0	1	0	1	1	0	1
1	0	1	0	0	1	1
1	1	1	0	1	0	1

TABLE 5.5 – Fonctions à deux entrées modélisées par des réseaux dont l'architecture est composée d'un seul lien.

Pour les deux premières fonctions, l'entrée reliée à la sortie est la première. Pour les deux suivantes, l'entrée reliée est la seconde. La dernière fonction peut être modélisée avec un lien vers la première ou la seconde entrée au choix. Les deux architectures introduites ci-dessus sont représentées dans la figure 5.23. Le réseau de gauche illustre la fonction f_1 tandis que celui de droite modélise la fonction f_4.

Le comportement observé est assez simple à expliquer. En effet, si nous analysons les deux premières fonctions du tableau, nous pouvons voir que celles-ci représentent les fonctions *identité* et *négation* pour la première entrée. Les deux fonctions suivantes font de même pour la seconde entrée. Il est donc évident qu'un seul lien partant de l'entrée dont dépend réellement la fonction est suffisant pour modéliser celle-ci. La fonction f_5, quant à elle, représente la fonction *unité*. Pour la modéliser, il suffit d'avoir un lien de poids 1 et un seuil négatif.

FIGURE 5.23 – Architectures des réseaux modélisant les fonctions logiques à deux entrées (1). Les architectures présentées ne contiennent qu'une seule connexion d'une des entrées à la sortie. Le réseau de gauche représente la fonction f_1 et l'entrée reliée à la sortie est la première. Au contraire, l'entrée reliée à la sortie dans la figure de droite est la seconde. La fonction modélisée est la fonction f_4.

Un grand nombre de fonctions logiques simples à deux entrées sont modélisées par des réseaux ayant la même architecture que ceux modélisant la fonction *ou* sur laquelle nous avons déjà travaillé (voir figure 5.24). Dans ces réseaux, chaque entrée est reliée à la sortie. Ces fonctions sont listées dans le tableau 5.6.

x_1	x_2	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}
0	0	1	0	0	0	1	1	1	0
0	1	0	1	0	0	1	1	0	1
1	0	0	0	1	0	1	0	1	1
1	1	0	0	0	1	0	1	1	1

TABLE 5.6 – Fonctions à deux entrées modélisées par des réseaux de trois neurones où les deux entrées sont reliées à la sortie.

Ces fonctions, bien qu'étant modélisées par des réseaux de même architecture, ont évidemment des poids et des seuils différents, ce qui permet de modéliser l'une d'entre-elles plutôt que l'autre.

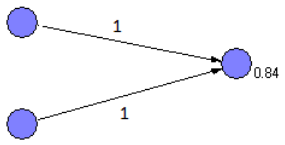

FIGURE 5.24 – Architectures des réseaux modélisant les fonctions logiques à deux entrées (2). Chacune des entrées possède une connexion vers la sortie. La fonction modélisée est la fonction *et* (fonction f_9). Nous avions déjà rencontré cette architecture lors de la modélisation de la fonction *ou*.

Si nous récapitulons les fonctions observées jusqu'à présent, nous obtenons un total de 14 fonctions. Toutes ces fonctions sont modélisées à l'aide de trois neurones. Il nous reste donc deux fonctions à analyser. Celles-ci sont les fonctions *xor* et *identité*. Les valeurs données à la sortie pour chacune des combinaisons d'entrées de ces fonctions sont données dans le tableau 5.7. La fonction f_{14} est la fonction *xor* et la fonction f_{15} est la fonction *identité*.

x_1	x_2	f_{14}	f_{15}
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

TABLE 5.7 – Fonctions *xor* et *identité*.

Contrairement aux fonctions analysées précédemment, ces deux fonctions ne peuvent pas être modélisées par des réseaux de trois neurones. En effet, lors de nos exécutions, nous obtenons des réseaux qui permettent d'obtenir les sorties attendues pour chaque combinaison d'entrées uniquement si nous ajoutons deux neurones intermédiaires. Les réseaux obtenus sont présentés dans la figure 5.25.

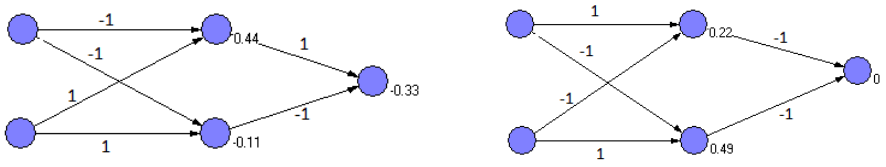

FIGURE 5.25 – Architectures des réseaux modélisant les fonctions logiques à deux entrées (3). La modélisation de ces fonctions nécessite l'introduction de neurones intermédiaires dans nos réseaux. Ces fonctions sont respectivement la fonction *xor* à gauche et la fonction *identité* à droite. Ces deux fonctions ne sont pas linéairement séparables.

La caractéristique commune à ces deux fonctions est qu'elles ne sont pas linéairement séparables. Nous avions déjà mis cette différence en évidence lors de l'analyse des neurones construits sur le modèle du perceptron. En effet, la modélisation de ces fonctions par ce modèle de réseaux nécessitait également l'ajout de neurones intermédiaires. Nous avions alors introduit la notion de décodeur et nous avions vu que le nombre maximum de neurones à ajouter était égal au nombre de combinaisons d'entrées donnant une sortie de 1.

Dans le cas de nos deux fonctions, le nombre de neurones à ajouter est exactement le nombre de sorties qui ont une valeur égale à 1. Nous allons voir si c'est toujours le cas en prenant des exemples de fonctions non linéairement séparables à trois entrées.

Fonctions logiques à trois entrées

Commençons par analyser les réseaux de neurones permettant de modéliser la fonction logique à trois entrées représentée dans le tableau 5.8. Cette fonction a été choisie de façon aléatoire. Nous nous sommes juste assurés de prendre une fonction qui soit bien non linéairement séparable.

Le meilleur réseau obtenu à l'aide de nos algorithmes pour la modélisation de cette fonction est composé de six neurones : les trois entrées, la sortie et deux neurones intermédiaires. Ce réseau est représenté dans la figure 5.26.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

TABLE 5.8 – Fonction logique à trois entrées non linéairement séparable.

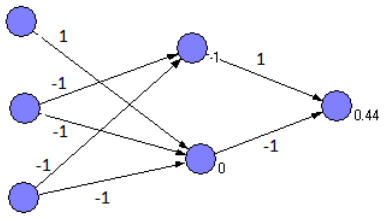

FIGURE 5.26 – Architectures des réseaux modélisant les fonctions logiques à trois entrées (1). La fonction modélisée est présentée dans le tableau 5.8. Elle est non linéairement séparable. Nous pouvons voir que nous avons besoin d'un nombre de neurones intermédiaires égal au nombre de combinaisons d'entrées qui donnent une sortie de 1.

De nouveau, nous pouvons observer que le nombre de neurones intermédiaires nécessaires à la modélisation de cette fonction est égal au nombre de combinaisons d'entrées ayant une sortie égale à 1. Il semblerait donc que le nombre de neurones à ajouter dans nos réseaux soit toujours le maximum possible.

Pour confirmer nos résultats, nous allons prendre une autre fonction non linéairement séparable. Celle-ci est semblable à la précédente. Nous lui avons simplement ajouté une combinaison d'entrées à laquelle correspond une sortie égale à 1. Cette fonction est représentée dans le tableau 5.9.

Lors de nos analyses, nous obtenons deux réseaux optimaux qui permettent de modéliser cette fonction. Ces réseaux sont composés de six neurones et sont représentés dans la figure 5.27. La modélisation de cette fonction nous permet de faire deux remarques importantes.

La première est que, cette fois, nous n'avons pas besoin du nombre maximal de neurones intermédiaires pour pouvoir modéliser notre réseau. En effet, les combinaisons d'entrées donnant une sortie de 1 sont au nombre de trois et nous avons seulement ajouté deux neurones à nos réseaux. Il semblerait donc que nos réseaux suivent bien la même loi que ceux construits sur le modèle de McCulloch-Pitts et du perceptron.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

TABLE 5.9 – Fonction logique à trois entrées. Cette fonction est identique à la fonction présentée dans le tableau 5.8 sauf pour une combinaison d'entrées. Elle est également non linéairement séparable.

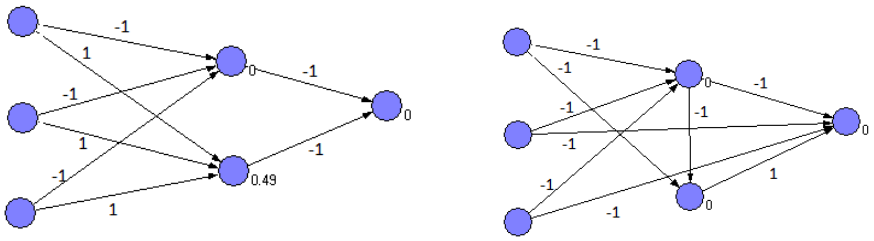

FIGURE 5.27 – Architectures des réseaux modélisant les fonctions logiques à trois entrées (2). La fonction modélisée est présentée dans le tableau 5.9. Elle est non linéairement séparable. Nous pouvons voir que nous avons besoin d'un nombre de neurones intermédiaires inférieur au nombre de sorties ayant une valeur égale à 1.

La seconde est que, pour la première fois, nous obtenons deux réseaux qui nous permettent de modéliser la même fonction. Nous avons donc un cas de dégénérescence. A première vue, il nous est impossible de départager nos deux réseaux. En effet, ils permettent tous deux d'obtenir une valeur de 1 pour le premier objectif. En ce qui concerne les objectifs 2 et 3, à savoir le nombre de connexions et le temps d'exécution, le réseau de gauche de la figure 5.27 a des valeurs de 0.73 et 0.67 tandis que celui de droite a des valeurs de 0.70 et 0.73. Cela signifie que le réseau de droite possède plus de liens que celui de gauche mais qu'il est quand même plus rapide que ce dernier. Il faudra donc faire un choix entre le nombre minimal de connexions et le temps d'exécution minimal.

Conclusion

Après avoir analysé ces fonctions logiques, nous pouvons tirer certaines conclusions. Il semblerait que toutes les fonctions logiques puissent être modélisées par nos réseaux de neurones. Pour ce faire, le nombre maximal de neurones intermédiaires nécessaires est égal au nombre de combinaisons d'entrées ayant une sortie égale à 1. De plus, il est possible de trouver différents réseaux qui permettent de modéliser parfaitement la fonction et qui sont optimaux pour le deuxième ou le troisième objectif.

5.2.2 Optimisation modulaire ou globale

Nous allons à présent observer le comportement de nos réseaux et les interactions apparaissant entre les différentes tâches, c'est-à-dire les différentes fonctions, à réaliser. Par exemple, existe-t-il des connexions qui lient des éléments (entrées, sorties) des deux fonctions si nous les modélisons en même temps ? Quelles sont ces connexions ? Quand apparaissent-elles ? Pour cela, nous allons prendre plusieurs exemples. Pour chacun d'entre-eux, nous reprendrons dans un premier temps les réseaux optimaux modélisant chacune des deux tâches séparément et nous les assemblerons. Dans un second temps, nous tenterons de modéliser les deux fonctions en même temps. Nous comparerons ensuite les résultats obtenus.

Grâce à ces analyses, notre objectif est d'obtenir des éléments de réponse sur l'organisation du cerveau. En effet, nous aimerions savoir s'il existe une zone du cerveau pour chaque tâche ou si certaines zones peuvent en réaliser plusieurs. L'apparition ou non de connexions entre les neurones des deux fonctions nous apportera un élément de réponse.

Commençons par prendre des fonctions logiques à deux entrées très simples, à savoir les fonctions *et* et *ou*. Nous connaissons le réseau optimal permettant de modéliser chacune de ces fonctions. Si nous les assemblons, nous obtenons le réseau de la figure 5.28. Ce réseau est le réseau optimal obtenu par optimisation modulaire. Si nous exécutons à présent nos algorithmes pour optimiser la modélisation de nos deux fonctions en même temps (optimisations globale) par un réseau de six neurones, c'est-à-dire les quatre entrées et les deux sorties, nous obtenons le même graphe optimal. Aucune connexion n'apparaît entre les éléments de la première fonction et de la seconde.

Remarque
Il est important de souligner que lorsque nous affirmons obtenir le même graphe optimal, cela signifie que nous obtenons la même architecture et les mêmes valeurs pour les poids des synapses du réseau. Les valeurs des seuils, quant à elles, peuvent légèrement varier puisque nous avons déjà vu qu'une fonction peut être modélisée à l'aide de plusieurs valeurs pour le seuil.

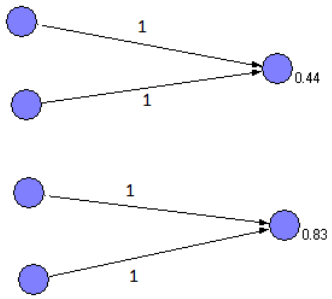

FIGURE 5.28 – Réseau modélisant les fonctions *ou* et *et*. Le réseau obtenu est le même si nous prenons les réseaux optimaux pour les deux fonctions et que nous les assemblons ou si nous optimisons les deux fonctions en même temps.

Si nous relançons nos algorithmes sur les mêmes fonctions pour des réseaux de taille cinq dans l'analyse modulaire, nous obtenons des graphes ne tenant pas compte des neurones intermédiaires. En les assemblant, nous obtenons le réseau représenté dans la figure 5.29. Nous obtenons de nouveau le même graphe si nous lançons nos algorithmes sur l'optimisation globale de nos deux fonctions par un réseau de taille dix. Remarquons une nouvelle fois que nos objectifs 2 et 3 sont importants dans ce type d'analyse puisqu'ils empêchent nos réseaux de posséder des connexions inutiles vers les neurones intermédiaires et d'être trop lents.

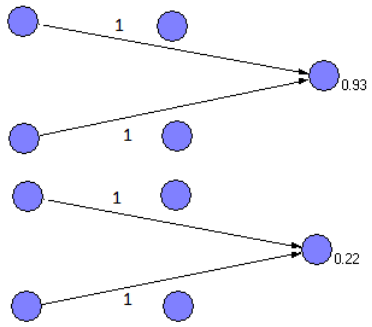

FIGURE 5.29 – Réseau optimal modélisant les fonctions *ou* et *et* à l'aide de 10 neurones. Le réseau optimal est le même pour l'optimisation modulaire et globale. De plus, les deux sortes d'optimisation permettent de reconnaître les neurones inutiles

Les résultats que nous venons d'obtenir ne sont pas étonnants. En effet, les seuls réseaux que nous venons d'analyser sont soit composés uniquement d'entrées et de sorties, soit composés d'entrées, de sorties et de neurones inutiles. Le fait d'ajouter des connexions liant des éléments des deux fonctions ne pourrait donc apporter aucune amélioration.

De même, si nous prenons une fonction composée de neurones intermédiaires et que nous voulons l'associer à une fonction qui n'en possède pas, nous n'aurons pas non plus de connexion entre les neurones des deux fonctions. Pour confirmer cette intuition, nous allons associer la fonction *xor* à deux entrées et la *négation* (une entrée). Dans ce cas, la fonction *négation* ne possède pas de neurones intermédiaires et nous nous attendons donc à observer le même comportement que précédemment. Le réseau obtenu par optimisation globale des deux fonctions est représenté dans la figure 5.30. Nous pouvons voir qu'il correspond bien à celui obtenu grâce à l'optimisation modulaire. Le comportement est donc celui auquel nous nous attendions.

Si nous prenons à présent deux fonctions à deux entrées possédant des neurones intermédiaires, à savoir les fonctions *xor* et *identité*, nous pouvons nous attendre à ce que les neurones intermédiaires utilisés par les deux fonctions soient les mêmes et à obtenir un réseau plus compact si nous optimisons nos deux fonctions en même temps. Le meilleur réseau obtenu par optimisation globale est représenté dans la figure 5.31.

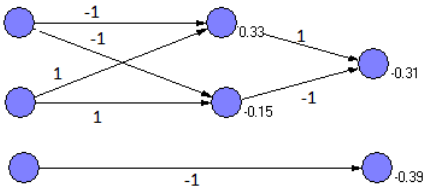

FIGURE 5.30 – Réseau optimal modélisant la fonction *xor* et la *négation*. Comme nous l'avions prédit, le réseau optimal est le même si nous optimisons chacune des deux fonctions séparément et que nous les assemblons ou si nous optimisons les deux fonctions en même temps.

Une fois de plus et contrairement à nos attentes, nous pouvons voir que ce réseau est le même que celui obtenu par optimisation modulaire. En effet, chacune des fonctions garde ses propres neurones intermédiaires. Comme ce résultat est assez inattendu, nous relançons notre algorithme génétique en supprimant un neurone pour voir s'il n'est tout de même pas possible de modéliser nos deux fonctions à l'aide d'un réseau plus petit. Le réseau optimal obtenu à la fin de cette exécution n'étant pas capable de modéliser correctement les deux fonctions, nous devons conclure qu'il n'est pas possible de modéliser nos deux fonctions sans utiliser les quatre neurones intermédiaires.

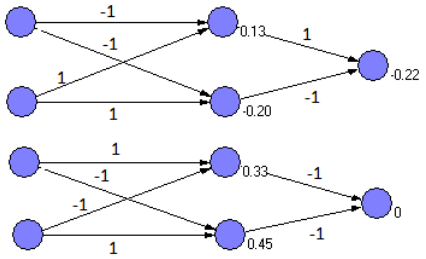

FIGURE 5.31 – Réseau optimal modélisant les fonctions *xor* et *identité*. Contrairement à ce que nous attendions, le réseau optimal est le même pour l'optimisation modulaire et globale.

Nous avons également lancé cette analyse sur des réseaux modélisant une des deux fonctions non linéairement séparables à deux entrées et une des deux fonctions non linéairement séparables à trois entrées analysées dans la partie précédente. Pour chacune de ces exécutions, nous avons obtenu le même résultat que pour la modélisation des deux fonctions à deux entrées. En effet, pour chacune de ces expérimentations, nous avons obtenu le même réseau final en utilisant l'optimisation modulaire et l'optimisation globale. Il n'était pas possible de modéliser les fonctions correctement avec des réseaux de plus petite taille.

En conclusion, il semblerait qu'il n'existe pas de connexion entre les neurones qui doivent réaliser nos différentes tâches. Cette observation tendrait à montrer qu'il existe une zone du cerveau pour chaque tâche. De plus, pour toutes les fonctions analysées précédemment, nous obtenons une redondance dans nos

réseaux. En effet, nous obtenons chaque fois le même réseau optimal alors que nous utilisons deux algorithmes génétiques et deux méthodes d'optimisation (modulaire ou globale). Cependant, les réseaux sur lesquels nous avons travaillé sont d'assez petite taille et il serait intéressant de relancer nos analyses pour des fonctions plus complexes et pour des réseaux composés d'un plus grand nombre de neurones. Etant donné le temps nécessaire à la convergence des réseaux de petite taille vers le réseau optimal (près de 200000 générations pour l'algorithme génétique à fitness pondérée), nous ne réaliserons pas ces analyses dans ce travail. De plus, nous n'avons travaillé que sur des réseaux modélisant des fonctions logiques. Il serait intéressant de travailler sur l'optimisation d'autres tâches pour voir si les observations obtenues sont les mêmes. Dans le chapitre suivant, nous réaliserons cette analyse sur des réseaux qui doivent contrôler des robots afin qu'ils suivent un gradient et qu'ils évitent des obstacles.

5.3 Conclusion

Ce chapitre a été consacré à l'analyse des algorithmes génétiques et des réseaux que nous avons implémentés. Dans un premier temps, nous avons analysé les paramètres de nos deux algorithmes génétiques de façon à les optimiser. Nous avons ensuite effectué une comparaison de ces algorithmes génétiques. Nous avons vu que l'algorithme génétique NSGA demande moins d'itérations que l'algorithme génétique à fitness pondérée mais que ces itérations prennent plus de temps.

Après cette première phase de test, nous nous sommes concentrés sur les réseaux optimaux obtenus lors de l'exécution de nos algorithmes. Nous avons vu que toutes les fonctions logiques étaient modélisables à l'aide de nos réseaux de neurones et que le nombre maximal de neurones intermédiaires suivait la même règle que les réseaux de neurones de McCulloch-Pitts et du perceptron. Nous avons également observé la redondance et la dégénérescence de nos réseaux. Nous avons vu que certaines fonctions étaient modélisables par des réseaux d'architectures différentes. Nous avons également observé que l'optimisation modulaire et l'optimisation globale menaient aux mêmes réseaux optimaux dans le cas où ces réseaux sont de petite taille et modélisent des fonctions logiques.

Chapitre 6

Application à la robotique

Pour terminer ce travail, nous allons utiliser nos réseaux de neurones pour contrôler des robots qui doivent réaliser des tâches simples. Nous utiliserons nos algorithmes génétiques pour faire évoluer nos réseaux de façon à ce que les robots acquièrent le comportement attendu. Ce chapitre a été réalisé à l'université de Bologne (campus de Cesena) sous la direction d'Andrea Roli.

Nous commencerons par donner quelques explications générales sur la façon dont les réseaux de neurones et les algorithmes génétiques peuvent être utilisés pour contrôler des robots. Nous expérimenterons ensuite une première tâche en nous basant sur un article de Beaumont [3]. Elle consistera à entraîner nos robots à suivre un gradient, c'est-à-dire à rechercher un optimum. Nous apprendrons ensuite à nos robots à éviter les obstacles. Dans un troisième temps, notre objectif sera de leur faire apprendre les deux tâches en même temps, ce qui nous permettra d'utiliser les algorithmes génétiques multi-objectifs implémentés à Venise. Nous terminerons ce chapitre en expliquant le fonctionnement du simulateur ARGoS utilisé à Cesena et en indiquant comment nous pourrions l'utiliser pour réaliser les tâches précédentes si nous avions plus de temps.

6.1 Concept général

Losque nous voulons qu'un robot réalise une tâche, nous avons le choix entre deux façons de procéder. La première façon est d'utiliser un robot auquel nous donnons toutes les règles à suivre pour agir de façon correcte. Ce robot est alors construit spécialement pour cette tâche. Dans ce cas, le robot est contrôlé par une carte électronique contenant un programme spécifiant les actions à effectuer. Une deuxième solution est de prendre un robot quelconque et de lui faire apprendre la tâche. Dans ce cas, le robot peut être contrôlé par des contrôleurs spécialisés ou différentes sortes de réseaux (neuronaux, booléens,...).

Nous allons nous intéresser dans ce travail à ce qui est appelé la robotique évolutionnaire. Ce domaine de recherche étudie le contrôle des robots par des réseaux de neurones en utilisant les algorithmes génétiques comme moyen d'apprentissage de la tâche à accomplir. Dans les années cinquantes, Turing suggéra pour la première fois que cette façon de procéder pourrait générer des systèmes

de contrôle efficace en robotique. Les premières expériences n'eurent cependant pas lieu avant le début des années nonantes. Depuis, la robotique évolutionnaire est devenue un domaine de recherche très étudié. En effet, un grand nombre de modéles de réseaux de neurones existent et beaucoup d'expérimentations différentes peuvent donc être réalisées à partir d'une même tâche.

Dans cette partie, nous allons commencer par expliquer comment des robots peuvent être contrôlés par des réseaux de neurones. Nous introduirons également l'importance des algorithmes génétiques dans le comportement de ces robots. Pour finir, nous donnerons plusieurs exemples d'expériences qui ont été réalisées en robotique évolutionnaire.

Un robot est composé de capteurs et d'actionneurs. Les capteurs lui permettent de percevoir son environnement (présence d'une lumière, d'un mur, déclinaison du sol,...). En fonction de ce qu'il perçoit, il peut réagir grâce aux actionneurs qui dirigent ses actions (déplacement des bras, roues, transmission d'un signal,...). Lorsqu'un robot est contrôlé par un réseau de neurones, les valeurs des neurones d'entrées de ce réseau sont données par les capteurs. Les neurones de sorties, quant à eux, contrôlent les actionneurs, et donc les actions du robot. En fonction des connexions existant à l'intérieur du réseau de neurones, le robot va réagir différemment à son environnement.

Nos algorithmes génétiques vont avoir le même rôle que précédemment. En effet, ils vont nous permettre de trouver un réseau de neurones permettant au robot de réaliser la tâche qui lui est demandée. Chaque individu de la population représente un réseau de neurones qui contrôle un robot. Selon les connexions existant entre les neurones du réseau, le robot réagit plus ou moins bien à son environnement et une fitness lui est attribuée en fonction de son comportement. Les méthodes de sélection, crossover et mutation sont ensuite utilisées pour obtenir la génération suivante de robots, plus performante que la précédente. Le processus est répété un certain nombre de générations afin d'obtenir un robot qui réalise le mieux possible la tâche demandée.

Un simulateur est nécessaire lorsque nous travaillons sur des problèmes pour lesquels nous souhaitons obtenir des résultats à tester sur des robots réels. En effet, le fait de travailler dans le monde réel introduit des paramètres physiques dont il faut tenir compte tels que les forces de frottements et d'autres forces physiques. Il nous faut donc un intermédiaire entre les sorties du réseau de neurones qui commande les actions du robot et ses véritables mouvements.

Différentes expériences ont été menées sur des robots qui devaient réaliser des tâches variées. Citons quelques exemples de façon à illustrer quelles genres de tâches peuvent être réalisées.

- La navigation sans collision [12]. Le robot doit apprendre à se mouvoir dans un environnement fermé (arène) en évitant les collisions avec des obstacles placés aléatoirement à l'intérieur de celui-ci.
- Le "Homing" [12]. Le robot doit se mouvoir dans un environnement et doit recharger sa batterie si nécessaire. Pour ce faire, il doit se trouver dans une partie précise de l'arène représentée par la présence d'une lumière

pour que le robot soit capable de la reconnaître. Il ne peut toutefois pas y rester car le fait de se trouver dans cette partie de l'arène un temps supérieur à celui nécessaire à sa recharge diminue sa fitness.

- La phototaxie et antiphototaxie [25]. Une lumière est placée dans un coin de l'arène. Dans un premier temps, le robot doit se déplacer vers la lumière. Ensuite, lorsqu'il entend un son brusque (par exemple un claquement de mains), il doit changer de direction et s'éloigner de cette lumière

D'autres expériences plus complexes ont également été menées pour observer le processus de coévolution entre une population de proies et de prédateurs ou l'évolution de la coopération et de l'altruisme à l'intérieur d'une population.

Après avoir brièvement introduit le concept général dans lequel nous nous plaçons, nous pouvons passer à l'expérimentation d'une première tâche.

6.2 Première expérience : suivre un gradient

Comme nous l'avons déjà annoncé, l'expérience que nous allons réaliser dans cette partie est basée sur une expérience présentée dans un article de Beaumont [3]. Il s'agit de l'apprentissage d'une tâche abstraite car l'environnement défini est artificiel.

Nous allons tout d'abord présenter le problème de façon générale. Nous décrirons ensuite l'environnement utilisé ainsi que les informations accessibles au robot. Nous donnerons également quelques précisions sur la façon d'utiliser ces informations dans nos réseaux. Nous présenterons ensuite les résultats obtenus et nous les analyserons.

6.2.1 Modélisation du problème

Le but de cette expérience est d'obtenir un robot capable de se mouvoir jusqu'au sommet d'une colline et de s'y arrêter. Autrement dit, notre robot doit apprendre à suivre un gradient. Nous commencerons par travailler sur des surfaces ne possédant qu'un seul sommet. Par la suite, nous pourrions envisager de travailler sur des surfaces plus complexes contenant plusieurs sommets de même hauteur ou de hauteurs différentes.

Environnement et robot

L'environnement de ce problème est une grille de 30 cases sur 30 qui représente la colline à monter. A chacune des cases de cette grille est assignée une hauteur particulière qui reste la même tout au long de l'entraînement du robot, c'est-à-dire tout au long de l'exécution de l'algorithme génétique. Une telle grille est représentée dans la figure 6.1. De même, nous définissons un ensemble de positions initiales qui vont servir à entraîner notre robot et qui resteront constantes tout au long de cet entraînement.

Le robot possède neuf capteurs qui lui permettent de connaître la hauteur de la case sur laquelle il se trouve et des huit cases qui l'entourent. Il n'a donc qu'une information locale de la surface sur laquelle il se déplace. Il est composé

de huit capteurs supplémentaires qui lui indiquent s'il se trouve à la frontière de la surface et de quelle côté est celle-ci. Le robot est également composé de quatre actionneurs qui lui permettent de se déplacer dans une des huit cases qui l'entourent ou de rester sur place. Comme ces actionneurs contrôlent le déplacement des robots, nous les appellerons également moteurs. Pour l'instant, nous considérons qu'il n'est pas possible pour le robot d'entrer en collision avec la frontière de la surface. Il n'est pas pénalisé s'il tente de se déplacer vers des cases qui n'existent pas. Ce mouvement lui est simplement interdit.

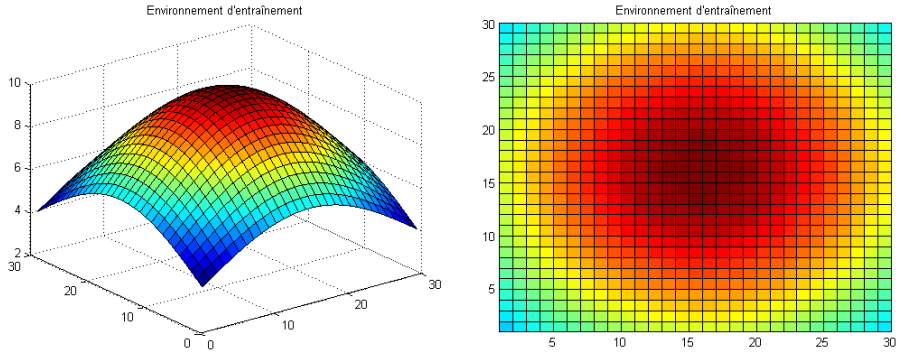

FIGURE 6.1 – Représentation de l'environnement d'entraînement du problème. La surface à gravir est décomposée en cases auxquelles on assigne une hauteur. L'environnement est constant tout au long de l'entraînement.

Après avoir introduit notre problème, nous allons à présent voir comment nous pouvons modéliser les informations reçues par les capteurs pour les introduire dans nos réseaux. De même, nous allons voir comment nous allons pouvoir utiliser nos sorties de façon à faire bouger notre robot. Nous donnerons également la façon de calculer la fitness de chaque robot en fonction de ses déplacements.

Implémentation

Pour pouvoir utiliser nos réseaux, nous avons besoin d'entrées binaires. Nous allons donc devoir transformer les hauteurs perçues par les capteurs du robot de façon à pouvoir insérer ces informations dans nos réseaux. Pour cela, l'état x_i des entrées correspondant aux neuf premiers capteurs est calculé à l'aide de la formule suivante :

$$x_i = \text{int}\ \frac{1.9(h_i - h_i^{min})}{h_i^{max} - h_i^{min}} \tag{6.1}$$

où h_i est la hauteur du i-ème capteur et où h_i^{min} et h_i^{max} représentent respectivement les hauteurs minimales et maximales parmi les neuf accessibles au robot. Cela signifie qu'une entrée est active si la différence entre la hauteur perçue et la hauteur minimale est supérieure à environ la moitié de la différence maximale de hauteur entre les 9 carrés observés. Pour les huit dernières entrées, leur état est 1 si le capteur correspondant a repéré une frontière et 0 dans le cas contraire.

Nous devons faire le même genre de travail pour relier les sorties du réseau de neurones aux moteurs. En effet, nos sorties binaires doivent exprimer le mouvement effectué par le robot. Pour cela, nous allons affecter les deux premières sorties au mouvement vertical et les deux dernières au mouvement horizontal. Le mouvement du robot suivra alors la règle du tableau 6.1.

Sortie a	Sortie b	Δ position
0	0	-1
0	1	0
1	0	0
1	1	$+1$

TABLE 6.1 – Mouvement effectué par le robot en fonction des sorties du réseau qui le contrôle. Les deux premières sorties dirigent le mouvement vertical et les deux suivantes le mouvement horizontal.

Cela signifie que la position d'un robot sera modifiée si les deux sorties qui sont assignées à la même direction ont la même valeur. Si cette valeur est 0, le mouvement est dans le sens négatif. Dans le cas contraire, le mouvement est positif.

Pour toutes les positions initiales de l'ensemble d'entraînement, le robot se déplace de 30 pas. Pour chacun de ses déplacements, nous retenons la hauteur des dix derniers pas qu'il effectue. Nous obtenons donc un tableau de valeurs possédant un nombre de ligne égal au nombre de positions initiales et un nombre de colonnes égal à 10. Nous prenons ensuite la moyenne de ce tableau que nous appelons fit. Nous calculons ensuite la fitness du robot en utilisant l'équation suivante :

$$fitness = \frac{fit - h_{min}}{h_{max} - h_{min}}$$

où $h_{min} = 3.93$ et $h_{max} = 9.98$ représentent les hauteurs minimales et maximales de la grille. Un réseau, et donc un robot, possède une fitness de 1 si, pour chaque position initiale de l'ensemble d'entraînement, il a atteint le sommet au plus tard au vingt-et-unième pas et qu'il est resté sur ce sommet par la suite.

Tous nos robots sont évalués à l'aide de cette fitness. Nous utilisons ensuite les méthodes de sélection, de crossover et de mutation de l'algorithme génétique pour obtenir la génération suivante de robots qui seront à leur tour évalués, sélectionnés, croisés et mutés pour obtenir la génération qui suit et ainsi de suite. Pour l'instant, la seule chose qui nous intéresse est d'obtenir un robot qui réalise correctement la tâche. Nous allons donc utiliser un algorithme génétique simple. Par la suite, nous pourrons envisager d'ajouter un deuxième objectif de façon à obtenir un réseau qui réalise la tâche et qui ne possède pas de connexions inutiles. Nous devrons alors utiliser un algorithme génétique multi-objectif.

Remarque
Nous ne pourrons pas utiliser la fonction objectif concernant le temps d'exécution du réseau pour cette partie du travail puisque nos entrées ne sont plus constantes. Nous ne chercherons donc pas à trouver un cycle et à voir le temps nécessaire à l'apparition de celui-ci.

Comme nos entrées sont complètement fixées par les capteurs à chaque pas, toutes les connexions qui arrivent à ces entrées sont inutiles. Il ne sert donc à rien de les faire muter au cours de l'algorithme génétique. Nous allons donc forcer la mutation à se faire sur la partie du réseau qui nous intéresse, à savoir les sorties et éventuellement les neurones de la couche cachée. Comme cela réduit fortement le nombre de connexions qui peuvent muter (les entrées sont au nombre de 17), nous allons diminuer de moitié notre taux de mutation et le fixer à 0.05. Nous le diminuerons de nouveau par la suite si nécessaire.

6.2.2 Résultats

Nous avons lancé notre algorithme génétique sur des réseaux de 21 neurones, à savoir les 17 entrées et les 4 sorties, ainsi que sur des réseaux de 26 et 31 neurones. Ceux-ci contiennent donc respectivement 5 et 10 neurones intermédiaires. Pour chacune de nos exécutions, l'environnement était celui présenté dans la figure 6.1. Nous utilisons dix positions initiales choisies aléatoirement pour entraîner nos robots. La figure 6.2 représente la fitness maximale de la population à chaque génération et pour nos trois tailles de réseaux. Nous pouvons voir que quel que soit le nombre de neurones qui composent nos réseaux, 100 générations sont suffisantes pour obtenir au moins un individu dont la fitness vaut 1.

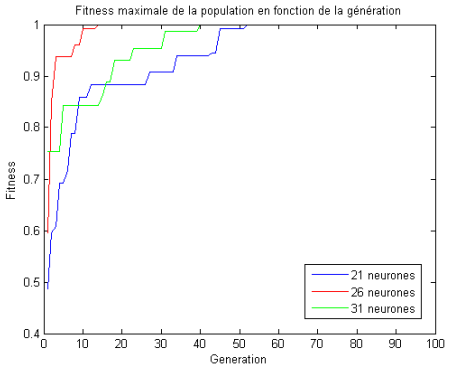

FIGURE 6.2 – Fitness maximale de la population à chaque génération pour des réseaux composés de 21, 26 et 31 neurones. L'algorithme converge vers au moins un individu dont la fitness est 1.

Si nous analysons le comportement des robots dirigés par le réseau optimal de chacune de nos trois exécutions (21, 26 et 31 neurones), nous pouvons voir qu'ils sont capables de trouver leur chemin jusqu'au sommet de la colline et de s'y arrêter à condition de partir d'une position qui ne se situe pas sur le bord de la surface. Ce comportement est représenté pour différentes positions initiales dans la figure 6.3. Les positions initiales sont indiquées en gris tandis que le mouvement du robot est représenté en noir.

Par contre, si nous partons d'une position se situant à la frontière de la surface, le comportement observé n'est pas celui attendu. Par exemple, si la position

initiale est le coin inférieur gauche de la surface, le robot n'est pas capable de se diriger de façon à suivre le gradient et il reste sur place.

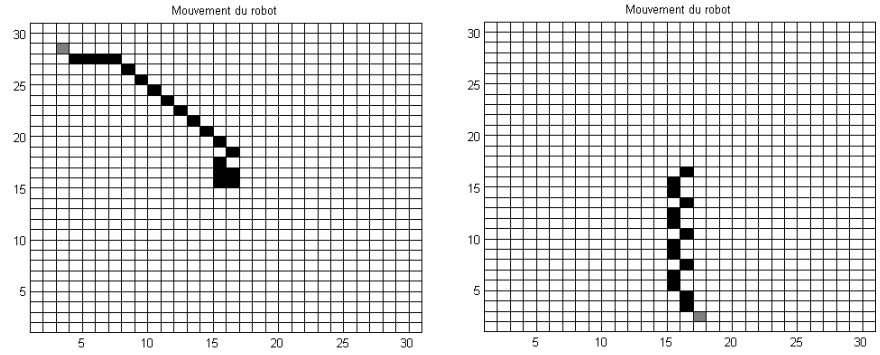

FIGURE 6.3 – Représentation du mouvement du robot. Les positions initiales sont des positions qui ne sont pas situées au bord de la surface. Elles sont représentées en gris. Le chemin suivi par le robot est représenté en noir. Nous pouvons observer que le robot est capable de trouver son chemin jusqu'au sommet de la colline et de s'arrêter. En d'autres mots, notre robot est capable de suivre un gradient.

Ce mauvais comportement est dû à la façon dont nous avons choisi l'ensemble des positions initiales pour entraîner nos robots. En effet, nous avons pris dix positions initiales qui ont été choisies aléatoirement. Parmi ces positions, aucune d'entre-elles ne représente une situation dans laquelle le robot démarre près de la frontière de la surface. Or, dans ce cas, au moins une entrée supplémentaire est active. Le robot, n'ayant jamais expérimenté cette configuration, ne sait pas comment y répondre et ne se déplace pas correctement.

Ajoutons huit positions initiales représentant des configurations où le robot démarre à la frontière de la surface (les quatre coins et le milieu de chaque côté) à nos dix positions initiales aléatoires et entraînons de nouveau nos robots à l'aide de ce nouvel ensemble. Dans ce cas, nous obtenons un mouvement correct du robot aussi bien à l'intérieur de la surface que sur sa frontière. Le mauvais comportement du robot était donc dû à un mauvais choix de l'ensemble d'entraînement lors de l'étape de modélisation du problème.

Remarque
Il n'est pas nécessaire d'utiliser les dix positions initiales choisies aléatoirement. En effet, le robot est capable d'apprendre à généraliser le mouvement à partir des huit positions initiales proches de la frontière. Nous continuerons toutefois à utiliser l'ensemble d'entraînement à dix-huit points pour la suite de nos analyses de façon à envisager tous les cas possibles dans l'apprentissage de notre robot.

Nous savons que notre robot est capable de réaliser la tâche demandée pour l'environnement d'entraînement. Nous allons à présent tenter de voir ce qui se passe si nous modifions légèrement ce dernier. Commençons par analyser le comportement du robot lorsque nous modifions la taille de la grille. Par la même occasion, nous devons augmenter le nombre de pas faits par le robot pour chaque

position initiale. Le comportement observé est représenté dans la figure 6.4 pour des grilles de 50 cases sur 50 et de 150 cases sur 150.

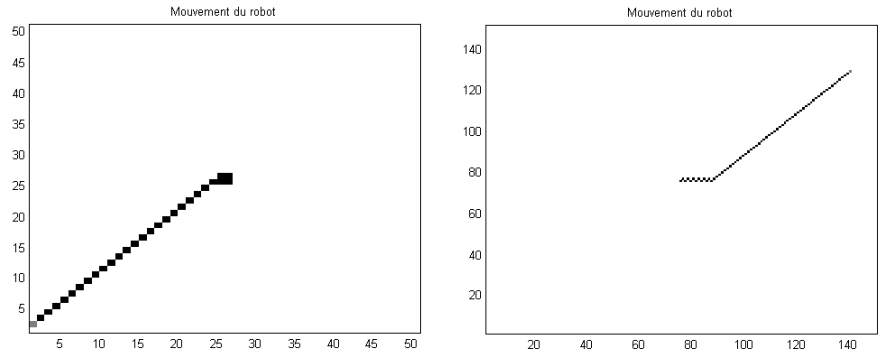

FIGURE 6.4 – Mouvement du robot pour des grilles de tailles différentes. Quelle que soit la taille de la grille, le robot est toujours capable de se diriger vers le sommet de la colline et de s'y arrêter.

Nous pouvons voir que quelle que soit la taille de la grille, notre robot est toujours capable d'atteindre le sommet, c'est-à-dire de suivre le gradient. La façon dont le réseau contrôle le robot est donc indépendante de la taille de la grille.

Analysons à présent ce qui se passe si nous changeons la place du sommet. Prenons par exemple l'environnement représenté dans la figure 6.5 et testons notre robot pour des positions initiales choisies aléatoirement à l'intérieur de celui-ci. Les résultats obtenus sont représentés dans la figure 6.6. Nous pouvons voir que le robot est toujours capable de se diriger vers le sommet.

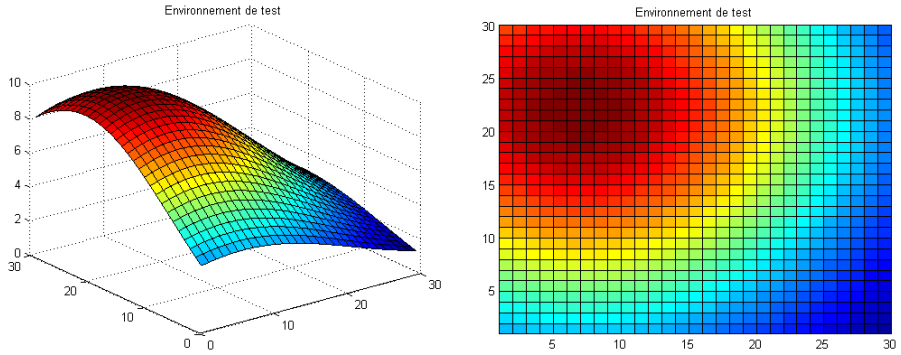

FIGURE 6.5 – Représentation de l'environnement de test du problème. La surface contient toujours un seul sommet mais celui-ci a changé de position. Cet environnement va nous permettre de tester notre robot de façon à savoir si celui-ci est capable de généraliser ce qu'il a appris sur la surface d'entraînement.

Suite à cette analyse, nous pouvons affirmer que nos trois exécutions de l'algorithme génétique nous permettent d'obtenir un réseau optimal capable de

contrôler correctement un robot dont la tâche est de suivre un gradient. De plus, ce contrôle et robuste. En effet, le robot est capable de se diriger correctement quelles que soient la taille de son environnement et la position du sommet à l'intérieur de celui-ci.

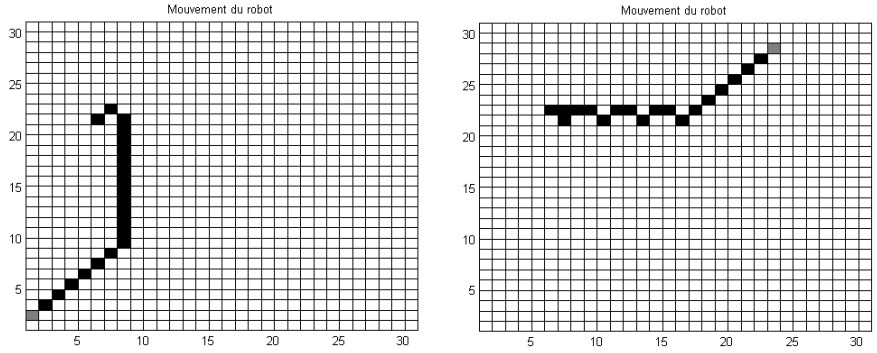

FIGURE 6.6 – Mouvement du robot sur l'environnement de test. Le robot se dirige bien vers le sommet de la colline dans ce nouvel environnement. Les positions initiales ont été choisies aléatoirement.

6.3 Deuxième expérience : éviter les obstacles

Nous allons à présent tenter d'apprendre à notre robot à éviter les obstacles. Cette tâche est plus complexe que la précédente. Comme pour celle-ci, nous commencerons par présenter le problème de façon générale. Nous donnerons ensuite quelques précisions sur la façon dont nous avons implémenté ses différents éléments. Nous analyserons ensuite les résultats obtenus.

6.3.1 Modélisation du problème

Comme nous l'avons déjà annoncé, nous allons tenter d'apprendre à nos réseaux à éviter les obstacles, c'est-à-dire à éviter les murs placés dans l'environnement sur lequel il se déplace. Nous allons à présent détailler cet environnement ainsi que la façon de représenter les obstacles. Nous verrons aussi quelles sont les informations auxquelles le robot a accès et la façon de calculer sa fitness.

Environnement et robot

Comme notre but final est d'entraîner notre robot à suivre un gradient et à éviter les obstacles simultanément, nous allons nous placer dans le même type d'environnement que précédemment de façon à pouvoir les relier par la suite. Nous travaillons donc toujours sur une grille de 30 cases sur 30.

Les murs sont représentés par des cases de la grille. L'ensemble d'entraînement n'est plus composé d'un seul environnement mais de plusieurs, et ce pour empêcher que le robot n'apprenne par coeur les positions des murs et évite ces positions. Ces environnements sont représentés dans la figure 6.7. Pour chacun

d'eux, nous prenons neuf positions initiales situées dans le bas de la grille et représentées en gris dans la figure précitée. A chaque itération, nous allons évaluer les compétences du robot pour chaque environnement et pour chacune des positions initiales dans ces environnements. Pour chacune de ces configurations, le robot se déplace de cent pas.

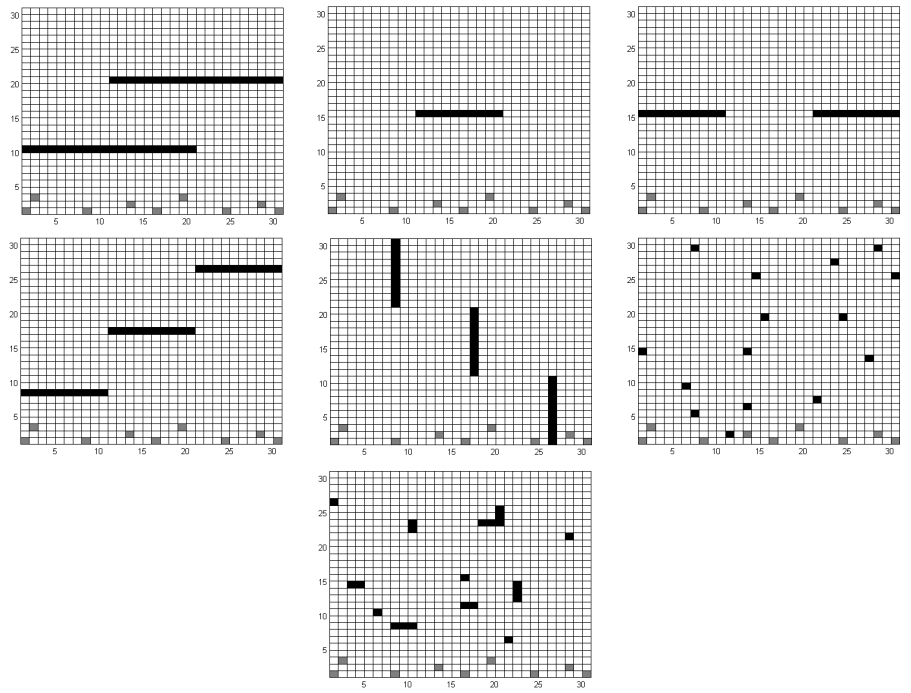

FIGURE 6.7 – Ensemble d'entraînement du robot. Les points initiaux sont représentés en gris et sont les mêmes dans chacun des environnements. Les murs sont représentés en noir. Il est important de prendre différentes configurations pour empêcher le robot de simplement apprendre les positions des murs de façon à éviter ces positions.

Les robots possèdent le même nombre de capteurs que précédemment. Les neuf premiers représentent toujours une évaluation de la hauteur en chacun des points de la grille et les huit autres représentent la présence ou non d'un mur dans les positions entourant le robot. Dans cette expérience, la hauteur des cases de la grille est 0 si la case est un mur et 1 sinon. Cela nous permet de donner une double information au robot sur la présence ou la non-présence d'un mur.

Implémentation

Nous avons toujours besoin d'utiliser des entrées binaires pour pouvoir utiliser nos réseaux comme contrôleurs des robots. La façon d'assigner une valeur aux huit dernières entrées, c'est-à-dire aux entrées représentant la présence ou l'absence d'un mur, reste la même. Pour rappel, leur état est 1 si le capteur correspondant détecte un mur et 0 sinon. Par contre, nous n'utiliserons pas l'équation 6.1 pour assigner un état aux neuf premières entrées car cela réduirait trop l'espace de ces entrées. En effet, la hauteur de la surface est 1 partout

sauf en présence d'un mur. Nous aurions donc des entrées dont l'état serait égal à 1 chaque fois que la case correspondante à cette entrée ne serait pas un mur. Pour éviter cette restriction, nous décidons d'assigner une valeur nulle aux entrées qui correspondent aux capteurs ayant détecté un mur. Les autres entrées, quant à elles, auront une valeur de 0 ou 1 qui leur sera assignée aléatoirement.

Remarquons que nos neuf premiers capteurs ne sont pas utiles dans cette expérience puisque la hauteur de la surface est partout la même excepté aux endroits où il y a un mur. Les huit derniers capteurs seuls seraient donc suffisant à la modélisation et à l'apprentissage de cette tâche. De plus, le fait d'éliminer ces entrées simplifierait le problème puisque les réseaux à optimiser seraient moins grands. Nous n'en ferons toutefois rien puisque notre but final est de relier nos deux tâches et d'obtenir des réseaux capables de les réaliser simultanément.

Les sorties de nos réseaux sont reliées aux moteurs du robot de la même façon que dans l'expérience précédente (voir tableau 6.1). Les deux premières sorties sont toujours reliées au mouvement vertical du robot et les deux dernières au mouvement horizontal. Le robot bouge si les deux sorties assignées à une direction ont la même valeur.

Pour toutes les positions initiales et ce dans chacun des environnements de l'ensemble d'entraînement, chaque robot se déplace de cent pas. Nous retenons ensuite la distance minimale entre les positions prises par le robot et le bord supérieur de la grille. Toutes ces distances minimales sont sommées et la distance totale est comparée à la distance maximale qui peut exister. La fitness est ensuite calculée de la façon suivante :

$$fitness = 1 - \frac{distance\_totale}{distance\_maximale}$$

Plus la distance totale par rapport à la frontière est petite, plus la fitness est grande. Cela signifie que nous considérons que notre réseau est bon s'il est capable d'atteindre l'autre côté de la grille en évitant les obstacles. Nous lui demandons donc non seulement d'éviter les obstacles mais aussi de traverser la grille. Nous ajoutons cette exigence de façon à ce que le robot ne se contente pas de rester sur place (ce qui est le moyen le plus simple pour lui d'éviter les obstacles) ou de tourner en rond.

6.3.2 Résultats

De nouveau, nous avons lancé notre algorithme génétique sur des réseaux de 21, 26 et 31 neurones. Le nombre maximal de générations est 10000. Comme nous l'avons déjà annoncé lors de l'introduction de cette tâche, celle-ci est beaucoup plus difficile à réaliser que la précédente. Aussi, nous ne sommes pas étonnés de voir que nous n'arrivons pas à converger vers des individus possédant une fitness de 1 (voir figure 6.8). Toutefois, lors de cet apprentissage, nous demandions non seulement aux robots d'éviter les obstacles mais aussi de traverser la grille. Nous allons donc analyser nos résultats de façon détaillée pour tenter de voir si le robot a appris quelque chose et ce qu'il lui reste à apprendre.

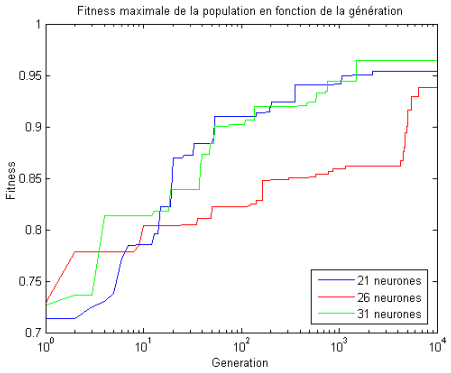

FIGURE 6.8 – Fitness maximale de la population à chaque génération pour des réseaux composés de 21, 26 et 31 neurones. Les 10000 générations de l'algorithme génétique ne permettent pas la convergence vers un individu possédant une fitness égale à 1.

Les résultats sont semblables pour les meilleurs réseaux des trois tailles. Nous pouvons voir que nos robots ont appris à éviter les obstacles de façon partielle. Un élément qui à première vue nous semble assez étonnant est que si nous suivons deux fois le trajet parcouru par un robot pour un environnement et un point de départ donnés, nous pouvons observer deux trajectoires différentes. De plus, le robot peut très bien éviter les obstacles dans l'une d'elles et entrer en collision avec le premier mur qu'il rencontre dans l'autre. La figure 6.9 illustre la situation énoncée ci-dessus.

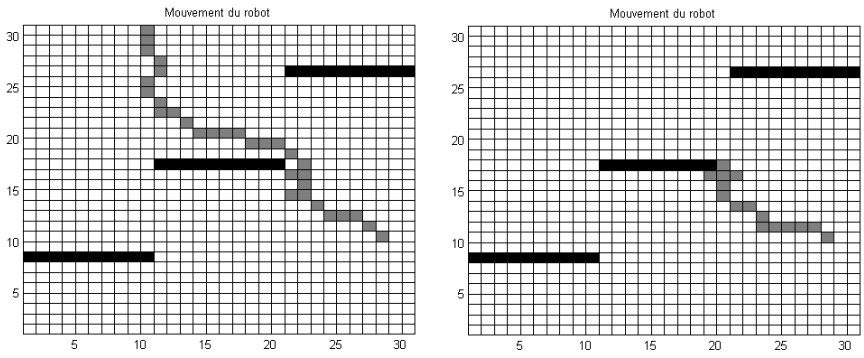

FIGURE 6.9 – Mouvement du robot. Le robot a appris la tâche mais pas totalement. Pour un même point de départ, nous avons des trajectoires différentes en fonction des valeurs données aux neuf premières entrées. Dans la figure de gauche, le robot est capable d'éviter le mur et d'atteindre le bord supérieur de la surface. Dans la figure de droite, le robot entre en collision avec le mur.

Cette différence de trajectoire peut s'expliquer par la présence des neuf premières entrées. En effet, nous avons assigné des valeurs aléatoires à celles de ces entrées qui ne se trouvaient pas près d'un mur. Les robots et les réseaux de neurones qui les contrôlent doivent donc faire face à un grand nombre de situations différentes. Ces neuf entrées à elles seules représentent en effet 9^2, c'est-à-dire

512 vecteurs d'états possibles. Et ces entrées doivent encore être couplées avec les huit entrées présentant la présence d'un mur ou non. Le nombre de situations qui doivent être gérées par nos réseaux est donc assez important.

Pour nous convaincre que la lenteur de convergence et donc la difficulté d'apprentissage de cette tâche est dûe à la présence de ces neuf premieres entrées, relançons notre algorithme en assignant une valeur nulle à chacune d'elles. De cette façon, elles n'auront jamais d'influence sur la trajectoire suivie par le robot. Dans ce cas, les résultats obtenus sont ceux attendus, à savoir le réseau a appris à éviter les obstacles et ce quelle que soit sa position initiale. Nous pouvons donc conclure que nos réseaux permettent l'apprentissage de cette tâche à condition de simplifier notre modélisation.

Dans l'exemple précédent, nous avions pu voir que nos réseaux optimaux étaient robustes par rapport à la taille de la grille et la position du sommet. Notre apprentissage est également robuste dans le cas de cette tâche. En effet, vu le nombre d'environnements et de points initiaux utilisés, il n'est pas possible que nos réseaux soient sur-entraînés. Nous allons à présent pouvoir passer à l'apprentissage de nos deux tâches en même temps.

6.4 Troisième expérience : suivre un gradient et éviter les obstacles

A la fin de cette troisième expérience, nous voudrions que nos robots soient capables de suivre un gradient tout en évitant les obstacles qui se dressent sur leur passage. Pour y arriver, nous allons expérimenter différentes méthodes :

- partir d'un réseau ayant appris l'une ou l'autre tâche lors des expériences précédentes et tenter de lui apprendre la seconde,
- repartir d'une population aléatoire de réseaux et apprendre les deux tâches en même temps.

Pour la première méthode, il nous faudra expérimenter ce qui se passe si nous partons d'un réseau capable de réaliser la première ou la seconde tâche. Il sera intéressant de voir si les résultats obtenus sont différents. En ce qui concerne la seconde méthode, elle nous permettra de travailler avec les algorithmes génétiques multi-objectifs implémentés à Venise. Nous allons commencer par décrire le problème et la façon dont nous combinons les deux expériences précédentes. Nous lancerons ensuite nos différents algorithmes et analyserons les résultats obtenus.

6.4.1 Modélisation du problème

Nous allons commencer par décrire ce qui est commun à nos méthodes, à savoir l'environnement, le type de robots utilisés et l'implémentation des interactions entre le robot et le réseau (capteurs-entrées et moteurs-sorties). Nous donnerons ensuite la fitness utilisée pour chacune d'entre-elles.

Environnement et robot

Cette expérience est une mise en commun des deux expériences précédentes. Il est donc évident que l'environnement et le robot vont posséder des caractéristiques semblables à ceux qui ont été utilisés lors de celles-ci. De ce fait, nous allons toujours travailler sur une surface qui est représentée par une grille de 30 cases sur 30.

Comme lors de l'apprentissage de la seconde tâche, l'ensemble d'entraînement n'est pas composé d'un environnement mais de plusieurs. Chacun de ces environnements représente une colline à gravir dont le sommet est situé au centre (voir figure 6.1). Chaque case possède donc une hauteur particulière. Sur chaque environnement, nous plaçons également une configuration d'obstacles à éviter. Les configurations sont les mêmes que celles utilisées pour la seconde tâche et sont représentées dans la figure 6.7. Lorsqu'une case contient un mur, nous considérons que sa hauteur est zéro. En ce qui concerne les positions initiales des robots, nous prenons des positions situées sur la frontière de la grille et des positions choisies aléatoirement comme pour la première expérience. Ces positions sont les mêmes pour chacun des environnements.

Le robot est exactement le même que celui utilisé lors des expériences précédentes. Il possède donc 17 capteurs et 4 moteurs. Les 9 premiers capteurs lui permettent de connaître la hauteur des cases qui l'entourent ainsi que la hauteur de celle sur laquelle il se trouve. Les 8 autres capteurs lui donnent une information concernant la présence ou non d'un mur pour la case correspondant à ce capteur. Les 4 moteurs, quant à eux, lui permettent de se mouvoir sur la grille.

Implémentation

A présent que nous avons de nouveau des hauteurs différentes pour les cases de la grille, nous allons réutiliser l'équation 6.1 pour assigner une valeur aux neuf premières entrées. Les huit autres sont toujours évaluées de la même façon, à savoir 1 si le capteur correspondant repère un mur et 0 sinon. Les sorties sont reliées aux moteurs de la même façon que pour les deux expériences précédentes.

Nous allons avoir une fitness différente en fonction de la méthode utilisée. Dans le premier cas, c'est-à-dire lorsque nous partirons d'un réseau qui permet déjà au robot de réaliser une des deux tâches, nous aurons une fitness semblable à celle utilisée lors de la première expérience. Les seules différences sont que le robot ne se déplace plus de 30 mais de 50 pas maximum et qu'il s'arrête dès qu'il entre en collision avec un mur. Nous retenons toujours les hauteurs des dix derniers pas pour chaque configuration et nous calculons toujours la moyenne de celles-ci. Nous normalisons ensuite cette moyenne pour obtenir la fitness. Si le robot entre en collision avec le mur avant d'avoir bougé de dix pas, il est pénalisé par une hauteur de zéro pour chaque pas non effectué.

Dans le second cas, nous allons avoir deux fonctions objectifs. La première est celle que nous venons de décrire qui aura pour but d'atteindre le sommet de la colline en tenant compte des obstacles. La seconde sera une pénalité ajoutée si

le robot entre en collision avec un mur. Cette pénalité est calculée d'une façon semblable à celle de la deuxième expérience à un changement près. En effet, comme le sommet est placé au centre de la surface et que nous souhaitons nous y arrêter, nous n'allons plus calculer la distance du robot par rapport à la frontière supérieure de la surface mais par rapport au centre de celle-ci.

Dans le cas où nous partirons d'une tâche acquise, la population sera initialisée de la façon suivante. Dix individus seront des copies conformes du meilleur réseau qui réalise cette tâche. Cinquante autres seront des mutations de ce réseau optimal et le reste de la population sera initialisé aléatoirement.

6.4.2 Résultats

Chacun de ces algorithmes a été lancé pour des réseaux de 21 neurones, à savoir les réseaux composés des 17 entrées et des 4 sorties. Pour chacune de ces exécutions, le nombre de générations est de 10000. Etant donné le temps nécessaire à leur exécution (2-3 jours) et la durée du stage à Cesena (1 mois), il n'était pas possible de lancer tous ces programmes pour les trois tailles de réseaux étudiées précédemment. Voyons à présent les résultats pour chacune de ces exécutions.

Tâche 1 ou 2 acquise

Analysons tout d'abord ce qui se passe si nous partons d'un réseau capable de réaliser une des tâches. La figure 6.10 représente la fitness maximale à chaque génération de l'algorithme génétique. Si la tâche acquise est la poursuite du gradient, nous obtenons une fitness maximale de 0.9595 à la fin de l'exécution. Dans l'autre cas, la fitness maximale est 0.9617 et est donc un peu meilleure.

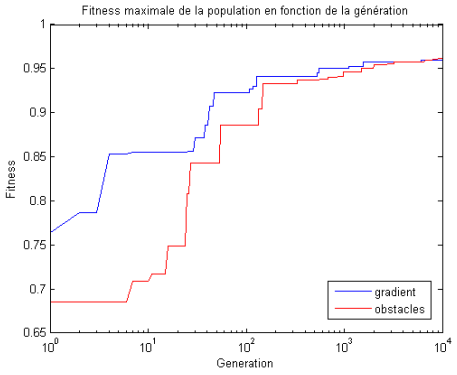

FIGURE 6.10 – Fitness maximale de la population à chaque génération. Les réseaux qui contrôlent les robots sont composés de 21 neurones. La fitness maximale obtenue si nous partons d'un réseau capable de réaliser la poursuite du gradient est de 0.9595 et celle obtenue en partant d'un réseau capable d'éviter les obstacles est de 0.9617. Nous pouvons voir que la population initiale a une meilleure fitness maximale si la tâche acquise est la poursuite du gradient.

Nous pouvons observer des comportements semblables pour les robots contrôlés par nos deux réseaux optimaux. En effet, ces robots sont capables de suivre

un gradient. Toutefois, ils ne s'arrêtent plus sur le sommet de la colline comme dans la première expérience mais se promènent ou s'arrêtent près de ce sommet. En ce qui concerne les collisions, les robots sont à présent presque toujours capables de s'arrêter avant la collision. Dans certains cas, ils n'ont cependant pas encore trouvé le moyen de contourner l'obstacle pour atteindre le sommet. Le comportement de ces robots est représenté dans la figure 6.11.

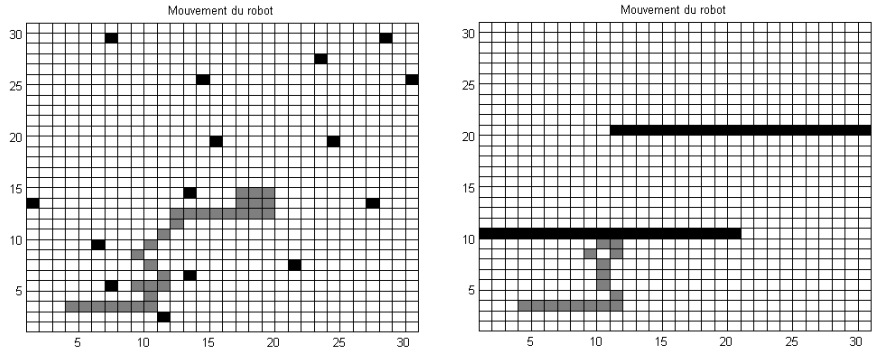

FIGURE 6.11 – Mouvement du robot. Dans la figure de gauche, le robot est capable d'éviter les obstacles et s'arrête sur une position proche du sommet. Sur la figure de droite, il est toujours capable d'éviter le mur mais il ne sait pas comment contourner l'obstacle pour s'approcher du sommet.

Nous pouvons tenter d'expliquer pourquoi les robots ne s'arrêtent plus au sommet. En effet, nous avons posé que la hauteur des cases de la grille sur lesquelles sont situés les murs est 0. Lorsque nous calculons la fitness, nous prenons la moyenne de la hauteur des dix derniers pas du robot. Dans le cas où celui-ci entre en collision avec un mur avant dix pas, la fitness des pas non faits est de 0. Nous normalisons ensuite la fitness en utilisant les hauteurs minimales et maximales. Or, lorsque nous réalisions cette étape pour la première expérience, la hauteur minimale était supérieure à trois et non pas zéro. Ce changement est peut-être à l'origine de cette modification dans le comportement du robot dans le sens où le fait de s'arrêter près du sommet est suffisant pour avoir une très bonne fitness.

Pour vérifier cette hypothèse, essayons de relancer une de nos exécutions en modifiant légèrement la fitness. Plutôt que de donner une valeur de zéro aux pas non effectués en cas de collision, donnons leur la valeur minimale de la surface. Si nous analysons les résultats obtenus, nous pouvons voir que le robot semble se rapprocher plus du sommet que dans les exécutions précédentes. Cependant, il ne s'arrête toujours pas sur celui-ci. Une autre solution possible serait d'augmenter le nombre de pas pris en compte dans le calcul de la fitness. Par manque de temps, cette proposition ne sera pas expérimentée dans ce travail.

Nous aimerions à présent savoir si le fait de connaître une tâche avant le début de l'exécution a avantagé la convergence ou si le réseau optimal est complètement différent de celui utilisé pour initialiser une partie de la population. Dans le cas où la tâche acquise est la poursuite du gradient, le réseau final permettant

de réaliser les deux tâches est assez semblable au réseau capable de réaliser la première tâche. En effet, la partie supérieure du réseau final a la même structure que celle du réseau réalisant la première tâche. Pour rappel, cette partie ne peut pas être modifiée lors de la mutation puisqu'elle n'a aucune influence sur le comportement du robot (les entrées sont modifiées en fonction des capteurs). Cela signifie que le réseau final est un descendant direct des réseaux réalisant la poursuite du gradient. De plus, la convergence vers le réseau final se fait assez rapidement puisque la fitness maximale est obtenue dès la 6039-ième génération. Au contraire, si la tâche acquise est la seconde tâche, c'est-à-dire si le robot est capable d'éviter les obstacles, le réseau final est complètement différent de celui de départ. Cela signifie que le réseau optimal n'est pas un descendant du réseau réalisant le seconde tâche mais qu'il provient d'un nouveau réseau introduit aléatoirement dans l'algorithme. De plus, la fitness maximale apparaît à partir de la génération 9131. Remarquons également dans la figure 6.10 que la population initiale a une meilleure fitness maximale si la tâche acquise est la poursuite du gradient. Il semblerait donc que le fait d'avoir acquis la première tâche soit plus intéressant que d'avoir acquis la seconde. Toutefois, nous n'avons lancé nos exécutions qu'une seule fois et cela n'est pas suffisant pour tirer des conclusions de façon définitive.

Algorithmes multi-objectifs

Nous allons à présent analyser les résultats obtenus avec les algorithmes multi-objectifs. Commençons par travailler sur l'algorithme génétique à fitness pondérée. Pour utiliser cet algorithme, nous avons dû donner des poids à chacune de nos fonctions objectifs. Comme nous souhaitions que notre robot puisse réaliser les deux tâches sans préférence pour l'une d'entre-elles, nous avons choisi de donner le même poids (0.5) à ces deux objectifs. La fitness maximale obtenue à chaque génération est représentée dans la figure 6.12.

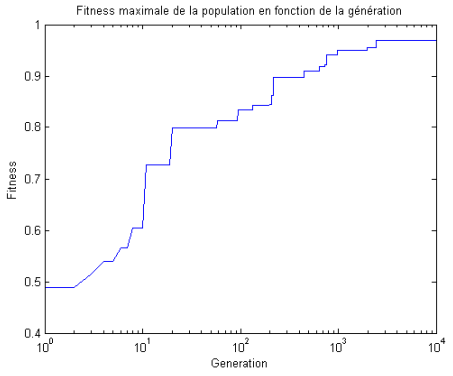

FIGURE 6.12 – Fitness maximale de la population à chaque génération. L'algorithme génétique utilisé est l'algorithme génétique à fitness pondérée. La fitness maximale à la fin de l'exécution est 0.9698. Cette fitness ne peut pas être comparée à celle des exécutions précédentes puisqu'elle est composée de deux fonctions objectifs.

A la fin de notre exécution, cette fitness maximale est de 0.9698. Cette fitness ne peut plus être comparée aux précédentes puisqu'elle est composée de deux

objectifs. Si nous calculons la valeur du premier objectif de façon à pouvoir le comparer à celui des deux exécutions précédentes, nous obtenons une valeur de 0.9432. Cette valeur est assez proche des valeurs obtenues pour les exécutions antérieures.

Comme nous pouvions nous y attendre au vu de la ressemblance du premier objectif de la fitness avec celui des deux exécutions précédentes, le comportement du robot contrôlé par le réseau optimal est semblable à celui des robots contrôlés par les réseaux optimaux de ces exécutions.

Si nous passons à présent à l'analyse de l'algorithme génétique multi-objectif, nous ne pouvons plus nous contenter de l'analyse d'un seul réseau. En effet, nous allons obtenir un ensemble de solutions Pareto-optimales et nous n'avons aucune raison de choisir une de ces solutions plutôt qu'une autre. Toutefois, pour éviter la multiplication des analyses, nous allons nous contenter d'analyser trois de ces solutions, à savoir les deux extrémités de la frontière Pareto optimale et une solution intermédiaire. La frontière Pareto optimale est représentée dans la figure 6.13. Les réseaux que nous allons analyser sont ceux dont les objectifs sont entourés en rouge.

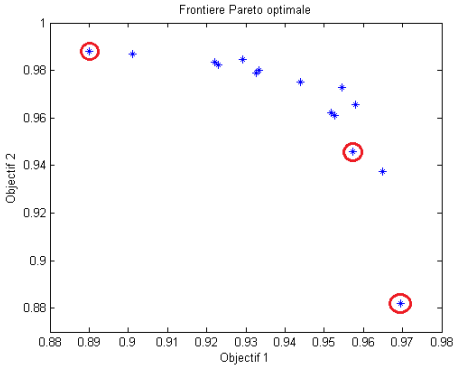

FIGURE 6.13 – Frontière Pareto optimale des solutions obtenues en utilisant l'algorithme génétique multi-objectif NSGA. Nous allons analyser le comportement des robots contrôlés par les réseaux dont les objectifs sont entourés en rouge. Ces réseaux représentent les deux extrémités de la frontière et une solution intermédiaire.

Commençons par analyser le réseau optimal qui possède la meilleure valeur pour le premier objectif. Cette valeur est 0.9696. La valeur du second objectif pour ce réseau est de 0.8819. Cette dernière valeur nous pousse à croire que le robot dirigé par ce réseau ne sera pas très bon pour éviter les collisions. Cependant, si nous analysons son comportement, nous pouvons voir que ce robot est tout à fait capable d'éviter les obstacles et de suivre un gradient. En réalité, le comportement observé est de nouveau assez semblable à celui observé pour les trois robots précédents. Cela n'a rien d'étonnant puisque la valeur du premier objectif est elle-même assez semblable à celles de ces robots.

Analysons à présent le réseau optimal qui possède la meilleure valeur pour le second objectif. Cette valeur est de 0.9882 et la valeur du premier objectif pour

ce réseau est de 0.89. L'analyse de ce réseau va nous permettre de mettre en évidence une erreur que nous avons commise dans notre façon de pénaliser les réseaux qui entrent en collision avec un obstacle. En effet, nous le pénalisons en regardant la distance qui le sépare du centre de la grille. Mais nous n'avons pas réfléchi à ce qui se passe dans le cas où l'obstacle lui-même se trouve au milieu de la grille. Dans ce cas, le réseau ne reçoit pas de pénalité pour la collision. Cette erreur entraîne que les réseaux n'ont pas appris à éviter tous les obstacles pour le réseau précité. De plus, la poursuite du gradient n'est pas très bonne. En conclusion, ce réseau n'est certainement pas celui qui dirige le robot de la façon souhaitée.

Remarque
Nous n'aurions pas pu remarquer cette erreur dans l'analyse des réseaux précédents car la valeur du second objectif était contrebalancée par celle du premier et n'avait pas autant d'influence que dans ce dernier réseau.

Terminons cette analyse en observant le comportement d'un robot dirigé par un réseau optimal intermédiaire. Les valeurs des deux fonctions objectifs sont respectivement 0.9573 et 0.9457. Si nous regardons son comportement plus en détail, nous remarquons qu'il est capable d'éviter les obstacles mais pas toujours de les contourner pour suivre le gradient. Cette poursuite du gradient, quant à elle, est moins précise que pour le premier réseau du front Pareto optimal que nous avons analysé. Nous préfèrerions donc utiliser ce dernier pour diriger notre robot.

Dans toutes nos analyses, nous avons pu remarquer les mêmes éléments concernant la robustesse de nos réseaux. En effet, nos robots sont capables de se diriger vers le sommet quelle que soit sa position. De plus, ce changement de position ne les empêche pas de continuer à éviter les obstacles. Par contre, si nous changeons la taille de la grille, ils sont toujours capables d'éviter les obstacles mais plus de suivre le gradient. Cette dernière observation est très étrange. En effet, il serait plus logique d'obtenir des difficultés lors du changement de position du sommet que lors du changement de la taille de la grille. Cela signifie dans un sens que le robot se sert de la taille de la grille pour avancer alors qu'il n'en a qu'une perception locale. Nous n'avons trouvé aucune explication à cette anomalie.

Optimisation modulaire ou globale

Nous venons d'analyser le comportement des robots dirigés par les meilleurs réseaux de neurones obtenus à l'aide de différentes optimisations globales. En effet, nous avons tenté d'apprendre les deux tâches l'une après l'autre ou en même temps à l'aide de nos algorithmes génétiques. Nous allons à présent observer les résultats obtenus lorsque nous prenons des réseaux capables de réaliser les tâches séparément et que nous les assemblons.

Pour cela, nous allons tout d'abord "nettoyer" les réseaux de neurones optimaux obtenus lors de l'optimisation de chacune des deux tâches. En effet, la première tâche ne dépend réellement que des 9 premières entrées, c'est-à-dire de la hauteur des cases qui entourent le robot et de celle sur laquelle il se trouve. De la même façon, la deuxième tâche ne dépend que des 8 entrées suivantes, à

savoir la présence d'un mur ou non dans les positions entourant le robot. Nous allons donc voir ce qui se passe si nous éliminons les liens entre les entrées qui n'ont pas d'importance et les sorties. Au cours de nos expériences, nous avons également permis la création de liens entre les sorties. Nous verrons si ceux-ci ont une influence dans la réalisation de nos deux tâches.

Pour la première tâche, nous obtenons toujours une fitness de 1 si nous enlevons les liens qui vont des 8 dernières entrées vers les sorties et si nous supprimons les liens entre les sorties. En ce qui concerne la seconde tâche, nous obtenons toujours une fitness de 1 si nous supprimons les connexions entre les 9 premières entrées et les sorties. Cette fois, nous ne pouvons cependant pas supprimer toutes les connexions qui lient les différentes sorties. En effet, nous devons maintenir un lien double entre les deux dernières sorties pour garder une fitness de 1.

Assemblons à présent nos deux réseaux "nettoyés" et analysons les résultats obtenus. Pour cela, nous créons un nouveau réseau à partir des connexions existant dans le premier réseau entre les 9 premières entrées et les sorties et de celles qui existent entre les 8 dernières entrées du second réseau et les sorties. Nous utilisons également le lien double existant entre les deux dernière sorties puisque celui-ci est essentiel à la réalisation de la seconde tâche.

La fitness du nouveau réseau est 0.7947 et est donc moins bonne que celle des réseaux obtenus à l'aide de nos optimisations globales. Cela n'a rien d'étonnant. En effet, nous avons légèrement changé l'objectif de nos deux tâches lorsque nous sommes passés à l'optimisation globale. D'une part, la hauteur minimale de la fitness a été modifiée. D'autre part, l'objectif de la seconde tâche est différent puisque nous ne devons plus traverser la grille mais nous arrêter en son centre. Nous pouvons donc conclure que dans ce cas, il vaut mieux optimiser les deux fonctions en même temps (ou l'une après l'autre) plutôt que de réaliser une optimisation modulaire.

6.5 Réflexion sur les résultats de nos expériences

Nous allons commencer par résumer ce que nous avons obtenu comme résultats lors de nos expériences. Nous attirerons ensuite l'attention sur quelques éléments qui nous ont marqués. Nous terminerons en donnant différentes pistes pour continuer notre travail.

6.5.1 Résumé

Lors de la première expérience, nous avons obtenu un réseau capable de diriger correctement un robot pour que celui-ci suive un gradient et s'arrête une fois le sommet atteint. Ce contrôle était robuste aussi bien pour la taille de la grille que pour l'environnement utilisé (place du sommet). Notre deuxième expérience nous a permis d'obtenir un robot qui peut éviter les obstacles à condition de simplifier légèrement notre implémentation. Le comportement du robot était également robuste. Enfin, en ce qui concerne la dernière expérience, nous avons obtenu un robot capable de suivre la direction du gradient sans s'arrêter au sommet et en évitant la plupart des obstacles. Il faudrait donc arriver à modifier

légèrement notre fitness pour que le robot s'arrête au sommet et laisser tourner l'exécution plus longtemps pour obtenir une tâche correctement réalisée.

6.5.2 Remarques

L'exécution de ces différentes expériences nous a permis de nous familiariser avec la robotique évolutionnaire. Un élément qui nous a particulièrement marqués est l'importance de l'ensemble d'entraînement. En effet, nous avons pu voir lors de la première expérience qu'un ensemble de points initiaux mal choisis ne permettait pas de suivre le gradient si nous partions du bord de la surface. Nous avons dû ajouter des points de ce bord dans l'ensemble d'entraînement. De même, lors de l'apprentissage du robot sensé éviter les obstacles, nous avons dû prendre des environnements assez différents de façon à ce que le robot ne se contente pas de mémoriser la place des obstacles.

La fonction de fitness a elle aussi une influence capitale. Comme nous avons pu le voir dans la dernière expérience, une petite modification de celle-ci peut entraîner des différences significatives dans le comportement du robot. En effet, nous avons légèrement modifié nos deux objectifs avec pour conséquences que le robot ne s'arrête plus au sommet et qu'il n'est pas pénalisé si la collision se passe au centre.

Nous devons donc prendre un soin tout particulier lors du choix de l'ensemble d'entraînement et de la fitness. En effet, ceux-ci vont déterminer le comportement final du robot. L'oubli d'un élément peut entraîner un comportement tout a fait inattendu. Par exemple, si nous demandons au robot d'éviter les obstacles sans lui spécifier qu'il doit bouger, il restera sur place.

6.5.3 Perspectives

Nous pouvons voir que nous avons obtenu des résultats satisfaisants mais qui peuvent être améliorés. Il serait donc intéressant de réfléchir à la façon de modifier les fonctions objectifs pour obtenir un robot qui réalise mieux les deux tâches ensembles. Nous pourrions également tenter de repartir de nos résultats et ajouter un critère pénalisant les connexions inutiles de façon à les voir disparaître.

De même, lors de la première expérience, nous avions envisagé de travailler sur des surfaces composées de plusieurs sommets de même taille ou de tailles différentes. Nous pourrions observer comment se comporte notre robot sur de telles surfaces. Nous pourrions également réfléchir à la façon de modifier notre fitness pour apprendre au robot à visiter les différents sommets avant de choisir sur lequel il doit s'arrêter.

Il serait également intéressant d'analyser en détail les réseaux obtenus lors de nos différentes expériences. Par exemple, nous pourrions voir si les différents réseaux obtenus pour la troisième expérience ont des points communs ou s'ils sont complètement différents. Nous pourrions également analyser la modularité de nos réseaux, c'est-à-dire voir si nous pouvons trouver des groupes de neurones qui sont fortement connectés entre eux et qui possèdent peu de connexions avec

le reste des neurones.

Nous pourrions également continuer notre travail en transformant nos expériences de façon à pouvoir utiliser le simulateur de Cesena pour analyser le mouvement des robots. Ce simulateur et la façon de l'utiliser seront expliqués dans la section suivante. Nous donnerons également quelques éléments de réponse sur la façon dont nous pourrions transformer nos problèmes pour pouvoir l'utiliser.

En conclusion, il existe un grand nombre de perspectives de travaux futurs qui pourraient se baser sur le travail que nous avons réalisé en robotique évolutionnaire lors du stage à Cesena.

6.6 Simulateur ARGoS

Nous allons terminer ce chapitre en présentant le simulateur utilisé à Cesena pour réaliser des expériences dont les résultats seront ensuite appliqués à des robots réels. Ce simulateur est donc assez technique et précis. Toutes les informations présentées à propos de celui-ci sont tirées de l'article de Pinciroli [23]. Nous envisagerons également la façon de transformer nos expériences pour pouvoir utiliser le simulateur sur celles-ci.

6.6.1 Description

L'utilisation d'un tel simulateur est centrale dans la robotique. En effet, elle permet d'obtenir des données expérimentales plus rapidement et à moindre coût qu'en utilisant des robots et des environnements réels. De plus, cela évite d'abîmer le matériel en y plaçant des robots mal contrôlés. Cela donne également la possibilité de travailler sur de grandes populations de robots, ce qui n'est pas toujours évident au vu du coût de ceux-ci. L'utilisation d'un simulateur est également très importante en robotique évolutionnaire. En effet, il n'est pas possible de calculer la fitness de chaque réseau en utilisant un robot réel. De fait, lors de nos expériences, nous avons créé une sorte de simulateur pour l'environnement abstrait utilisé.

Le simulateur ARGoS possède une architecture modulaire représentée dans la figure 6.14. Chaque élément est défini par un module différent et ces modules interagissent entre-eux. L'avantage de cette architecture est qu'elle permet de modifier une partie du code sans devoir réécrire les autres parties. Les modules du simulateur sont codés en C++ et celui-ci fonctionne sous Linux et Mac.

Les différents modules qui composent ce simulateur sont détaillés ci-dessous.

- L'espace 3D simulé est le centre de l'architecture. Il contient toute l'information sur l'état de la simulation. Il est composé de plusieurs entités. Chacune d'entre-elles enregistre l'information concernant un aspect spécifique de la simulation. Par exemple, un robot est représenté dans l'espace simulé par une entité composée, c'est-à-dire une entité contenant d'autres entités telles que l'entité qui enregistre l'information spatiale du robot ou l'entité qui stocke son comportement en fonction du contrôleur.

- Les capteurs et les actionneurs sont les éléments qui permettent au robot d'interagir avec son environnement. Les capteurs sont des modules qui lisent l'état de l'espace simulé. Ils exploitent la structure de cet espace en ne lisant que les informations contenues dans les entités qui les intéressent. Les actionneurs, quant à eux, écrivent dans les entités du robot et influencent donc son comportement.
- Les contrôleurs sont les modules qui décident de la façon dont le robot réagit à son environnement. Les capteurs leur transmettent l'information lue dans les entités. En fonction de celle-ci, les contrôleurs envoient aux actionneurs des informations sur la façon dont le robot doit se comporter.
- Les moteurs physiques sont les modules qui mettent à jour l'entité contenant l'information spatiale du robot. Cette mise à jour ne peut pas se faire directement par les actionneurs car l'environnement du robot est l'environnement physique réel et nous devons donc tenir compte de certains paramètres physiques. Par exemple, si l'actionneur donne l'ordre de mettre la vitesse du robot à zéro, cela signifie qu'il veut que le robot s'arrête. Or, dans la réalité, si nous bloquons les roues du robot, il va continuer à avancer pendant un certain temps. Dans ce cas, le moteur physique devra non seulement mettre la vitesse à zéro mais ralentir le robot pour qu'il s'arrête. Les moteurs physiques représentent donc une façon de transformer les mouvements artificiels commandés par les actionneurs en mouvements réels dans l'espace simulé.
- Les modules de visualisation lisent l'état de l'espace simulé et nous en donnent une représentation.

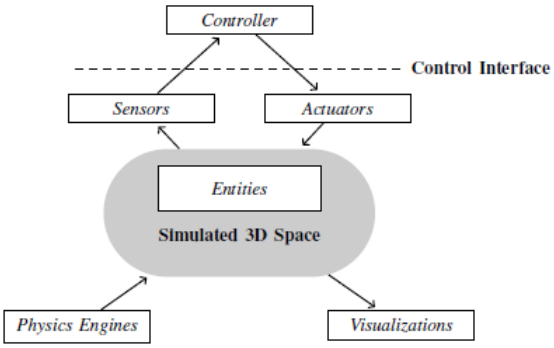

FIGURE 6.14 – Architecture du robot ARGoS. Les modules pouvant être définis par l'utilisateur sont représentés par les boîtes blanches. Source : [23] .

Le simulateur ARGoS permet de travailler sur différents types de robots et sur un nombre assez élevé de ceux-ci. De plus, l'exécution de la simulation se fait en parallèle ce qui permet au simulateur d'être assez rapide. Une dernière caractéristique de ce simulateur est qu'il permet de partitionner l'espace simulé en sous-espaces qui peuvent être dirigés par des moteurs physiques différents. Cela permet, par exemple, de travailler à la fois sur des robots volants et des robots munis de roues en leur assignant une dynamique adaptée à leurs besoins.

6.6.2 Utilisation sur nos expériences

Il serait possible d'utiliser le simulateur ARGoS sur nos expériences à condition d'apporter quelques modifications à leur modélisation et de coder quelques modules pour les insérer dans ce simulateur. Tout d'abord, nous devrions transformer l'espace discret sur lequel nous avons travaillé (grille de 30 cases sur 30) en espace continu. De même, plutôt que de travailler sur une surface dont les points ont des hauteurs différentes, nous pourrions travailler sur une surface ayant des intensités lumineuses différentes. Cela nous permettrait d'implémenter la poursuite du gradient sans devoir tenir compte du comportement physique du robot en fonction de la pente de la surface. Nous devrions également réfléchir à la façon dont nous implémenterions nos obstacles sur cette nouvelle surface.

En ce qui concerne les capteurs, nous pourrions les modéliser de façon assez similaire à celle utilisée lors de nos expériences. Les moteurs, au contraire, devraient être complètement modifiés. En effet, les robots encodés dans le simulateur ne sont pas capables de se mouvoir dans toutes les directions. Il doivent d'abord subir une rotation avant de pouvoir se déplacer. Il nous faudrait donc réfléchir à la façon d'introduire cette rotation dans notre expérimentation.

Enfin, nous devrions coder un contrôleur qui serait un réseau de neurones. Pour cela, il nous suffirait de traduire notre code matlab en langage C++. Comme nous l'avons laissé entrevoir, nous utiliserions les robots et les moteurs physiques qui sont déjà implémentés dans le simulateur pour réaliser cette expérience.

Une autre possibilité à envisager serait de relier directement le simulateur à notre algorithme génétique implémenté en matlab. De cette façon, le simulateur calculerait la fitness de chaque individu et la renverrait à notre algorithme génétique pour que celui-ci réalise les opérations de sélection, de crossover et de mutation. Nous obtiendrons ainsi des réseaux optimaux que nous pourrions testés sur des robots réels.

En conclusion, il serait intéressant de transformer nos expériences sur des tâches abstraites en expériences sur un environnement physique réel. Cela serait possible grâce au simulateur présenté ci-dessus.

6.7 Conclusion

Ce chapitre nous a permis d'appliquer nos implémentations à la robotique évolutionnaire. Nous avons tout d'abord introduit les concepts généraux qui y sont liés. Nous avons ensuite réalisé trois expériences abstraites. La première consistait à apprendre à nos robots à poursuivre un gradient, la deuxième à leur apprendre à éviter les obstacles et la troisième à leur apprendre les deux tâches à la fois. Cette dernière expérience nécessitait l'utilisation des algorithmes génétiques multi-objectifs implémentés à Venise. Les résultats obtenus pour les deux premières expériences étaient très satisfaisants. En effet, nos robots étaient parfaitement capables de suivre un gradient ou d'éviter les obstacles. En ce qui concerne les résultats de la troisième expérience, ils pourraient être améliorés.

A la suite de ces expériences, nous avons introduit l'utilisation du simulateur ARGoS. Nous avons également réfléchi à la façon de transformer nos expériences abstraites en expériences réelles de façon à pouvoir utiliser ce simulateur pour les réaliser. Nous avons vu que cette transformation était possible et que des expériences réelles pourraient donc être réalisées à l'aide de nos implémentations.

Conclusions et perspectives

Tout au long de ce mémoire, nous avons étudié la modélisation et l'apprentissage des réseaux de neurones artificiels. Nous avons tout d'abord introduit les fondements biologiques qui inspirent ces réseaux et nous en avons présenté deux modèles particuliers. Nous nous sommes ensuite intéressés à leur apprentissage. Dans un premier temps, nous avons introduit l'apprentissage classique. Comme nous voulions optimiser les poids et seuils des réseaux mais aussi leur architecture, nous avons décidé dans un second temps d'utiliser les algorithmes génétiques multi-objectifs lors de cet apprentissage. Nous sommes ensuite passés à l'implémentation d'un modèle de réseaux et de deux algorithmes génétiques multi-objectifs dont un à fitness pondérée et l'autre de type NSGA. Deux phases d'analyse ont suivi cette implémentation. La première a permis d'obtenir les valeurs des paramètres de nos algorithmes génétiques permettant la convergence pour un nombre minimal de générations. La seconde phase nous a permis d'étudier la dégénérescence et la redondance de nos réseaux. Nous avons terminé notre travail en appliquant les réseaux et les algorithmes génétiques implémentés au domaine de la robotique évolutionnaire.

Nous allons à présent revenir sur les principaux résultats obtenus au cours de ce travail. Nous allons commencer par donner les résultats concernant les algorithmes génétiques utilisés lors de l'apprentissage. Nous donnerons ensuite les résultats concernant les réseaux optimaux et la relation avec la robotique.

Tout d'abord, nous avons vu que nous devons utiliser des méthodes d'optimisation multi-objectif pour l'apprentissage de nos réseaux. En effet, si nous n'introduisons pas des pénalités sur le nombre de connexions et le temps d'exécution, nous obtenons des réseaux optimaux qui possèdent des connexions inutiles ou qui sont lents. Nous avons ensuite analysé les paramètres de nos algorithmes génétiques de façon à obtenir une combinaison de ceux-ci permettant une convergence en un nombre minimum de générations. Enfin, nous avons vu que l'algorithme génétique à fitness pondérée demande plus de générations que l'algorithme génétique multi-objectif NSGA pour converger mais que le temps d'exécution d'une de ses générations est beaucoup plus petit.

Pour rappel, le modèle de neurones implémenté est un modèle simplifié du perceptron. Il permet de modéliser toutes les fonctions logiques. De plus, le nombre de neurones intermédiaires à ajouter est plus petit ou égal au nombre de sorties qui ont une valeur de 1. Avec ce modèle, il est possible de trouver des réseaux de neurones structurellement différents qui modélisent la même fonction. Par contre, l'optimisation modulaire et globale des différentes tâches à réaliser nous

ont menés aux mêmes réseaux optimaux.

Il est possible d'utiliser nos réseaux de neurones pour contrôler des robots qui doivent réaliser des tâches abstraites. Il serait également possible de les utiliser afin de réaliser des tâches concrètes à condition d'utiliser un simulateur et un environnement adapté. Nous avons vu que la définition de la fitness et de l'ensemble d'entraînement est très importante et doit être pensée sérieusement pour obtenir le bon comportement du robot.

Attirons finalement l'attention du lecteur sur quelques limites de notre travail et sur quelques perspectives de travaux futurs. Tout d'abord, notre modèle a des entrées et des sorties binaires. Il ne nous permet donc pas de modéliser des fonctions continues. De plus, lors de l'application à la robotique, cette limite nous a obligé à trouver un moyen de transformer les informations reçues en entrées binaires et nos sorties binaires en mouvements exécutables par le robot. Il serait envisageable d'implémenter d'autres modèles de réseau de neurones, plus complexes ou possédant des fonctions d'activation continues.

Ensuite, lors de nos analyses sur l'optimisation modulaire et globale, nous avons dû nous limiter à l'analyse de réseaux de petite taille étant donné que le nombre de générations nécessaires à la convergence de tels réseaux étaient déjà assez important. Il serait donc intéressant de continuer cette analyse sur des réseaux possédant plus de neurones. Il serait également intéressant de réaliser cette analyse sur la modélisation de tâches autres que les fonctions booléennes.

Enfin, nous n'avons pas eu le temps de réaliser une analyse complète des réseaux optimaux obtenus pour les différentes expériences réalisées en robotique. Il serait intéressant de réaliser cette analyse. Une autre perspective serait de relier nos codes au simulateur ARGoS de façon à transformer nos tâches abstraites en tâches réelles et de réaliser des expériences sur ces dernières.

Bibliographie

[1] *Le petit Larousse 2010.* Larousse, Paris, 2010.

[2] A. Banerjee. Background : The biophysics of neuronal activity. http ://www.cise.ufl.edu/ arunava/Teaching/Lectures-CN/neuroelectronics.pdf, 2004. [En ligne ; visité le 16/03/2011].

[3] M.A. Beaumont. Evolution of optimal behaviour in networks of boolean automata. *Journal of Theoretical Biology*, 165 :455–476, 1993.

[4] U. Bodenhofer. *Genetic Algorithm : Theory and Applications (Lecture Notes, second edition).* Johannes Kepler universität, Linz, Austria, 2001-2002.

[5] J.C. Bongard. Spontaneous evolution of structural modularity in robot neural network controllers. *Genetic and Evolutionary Computation Conference (GECCO 2011), Dublin, IR*, 2011.

[6] J. A. Bullinaria. Using evolution to improve neural network learning : pitfalls and solutions. *Neural Computing and Applications*, 16(3) :209–226, 2007.

[7] J. Cronin. *Mathematical Aspects of Hodgkin-Huxley Theory.* Cambridge University Press, Cambridge, 1987.

[8] A. Pratap S. Agarwal Deb, K. and T. Meyarivan. A fast elitist multi-objective genetic algorithm : Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2) :182–197, 2002.

[9] K. Deb. Multi-objective evolutionary algorithms : Introducing bias among pareto-optimal solutions. *Advances in Evolutionary Computing*, Part I :263–292, 2003.

[10] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* John Wiley & Sons Ltd, Chichester, 2008.

[11] A. H. F. Dias and J. A. de Vasconcelos. Multiobjective genetic algorithms applied to solve optimization problems. *IEEE Transactions on Magnetics*, 38(2) :1133–1136, mar 2002.

[12] D. Floreano and L. Keller. Evolution of adaptive behaviour in robots by means of darwinian selection. *PLoS Biol*, 8(1) :e1000292, 01 2010.

[13] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization : formulation, discussion and generalization. In *Proceedings of the ICGA-93 : fifth international conference on genetic algorithms.* July 1993.

[14] Gerstner and Kistler. *Spiking Neuron Models. Single Neurons, Populations, Plasticity.* Cambridge University Press, 2002.

[15] D E Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[16] A. Hardy. *Syllabus du cours de "Statistiques"*. Facultés universitaires Notre-Dame de la Paix, Namur, 2008-2009.

[17] R. Hendrickx. Les algorithmes génétiques : Théorie et applications. Master's thesis, FUNDP (NAMUR), 2011.

[18] D. W. Coit Konak, A. and A. E. Smith. Multi-objective optimization using genetic algorithms : A tutorial. *Reliability Engineering and System Safety*, 91 :992–1007, 2006.

[19] D. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, Cambridge, 2005.

[20] D. Matthieu. Les algorithmes génétiques. http ://www.a525g.com/intelligence-artificielle/algorithme-genetique.htm, 2002. [En ligne ; visité le 28/08/2011].

[21] P. Peretto. *An introduction to the modeling of neural networks*. Aléa Saclay. Cambridge University Press, Cambridge, 1992.

[22] S. Piazza. Algorithmes génétiques et leurs applications aux réseaux de neurones. Master's thesis, FUNDP (NAMUR), 2003.

[23] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Timothy Stirling, Álvaro Gutiérrez, Luca Maria Gambardella, and Marco Dorigo. Argos : a modular, multi-engine simulator for heterogeneous swarm robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, pages 5027–5034. IEEE Computer Society Press, Los Alamitos, CA, September 2011.

[24] R. Rojas. *Neural networks : A systematic introduction*. Springer, Berlin, 1996.

[25] A. Roli, M. Manfroni, C. Pinciroli, and M. Birattari. On the design of Boolean network robots. In C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcázar, J. Merelo, F. Neri, M. Preuss, H. Richter, J. Togelius, and G. Yannakakis, editors, *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 43–52. Springer, Heidelberg, Germany, 2011.

[26] N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3) :221–248, 1994.

[27] J.-J. Strodiot. *Syllabus du cours d'"Optimization and control"*. Facultés universitaires Notre-Dame de la Paix, Namur, 2009-2010.

[28] O. Sporns Tononi, G. and G. M. Edelman. Measures of degeneracy and redundancy in biological networks. *Proceedings of the National Academy of Sciences, USA*, 96 :3257–3262, mar 1999.

[29] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9) :1423–1447, 1999.

[30] R. Younes. Chapitre 3 : Les réseaux de neurones. http ://www.ryounes.net/cours/chapitre%203%20RN.pdf, 2005. [En ligne ; visité le 22/04/2011].

Annexes

Nous présentons dans ces annexes les codes qui ont servis à réaliser les analyses du chapitre 5. Pour chaque algorithme, nous décrivons brièvement les différentes fonctions présentées. En ce qui concerne les codes du chapitre sur la robotique, ils sont disponibles sur demande.

Algorithme génétique à fitness pondérée

Cette partie contient les codes qui permettent d'optimiser nos réseaux à l'aide de l'algorithme génétique à fitness pondérée. Les codes présentés sont le programme principal, la fonction réalisant l'algorithme génétique, la fonction de calcul de la fitness et la fonction utilisée pour trouver un cycle dans nos réseaux.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auteur : Nicolay Delphine
% Date : 20/10/2011
% But: trouver un reseau optimal permettant de modeliser une fonction
%      particuliere (ici la fonction logique ou)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear
clc

% Initialisation de la forme du reseau
nb_input = 2;
nb_output = 1;
nb_int = 0;

% Initialisation du nombre de combinaisons d'entrees (2^nb_input), de leurs
% valeurs et de la sortie attendue pour chacune d'entre-elles
c = 4;
entree = zeros(nb_input,c);
entree = [0,0,1,1;0,1,0,1];
cible = zeros(nb_output,c);
cible = [0,1,1,1];

% Appel à l'algorithme genetique
[G,fit] = ag(nb_input,nb_output,nb_int,entree,cible);

disp('Le réseau optimal est')
```

```
G(:,:,200)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auteur : Nicolay Delphine
% Date : 20/10/2011
% But: optimiser le reseau pour modeliser une fonction
%      particuliere
%
% IN : nb_input = nombre d'entrees
%      nb_output = nombre de sorties
%      nb_int = nombre de neurones intermediaires
%      entree = matrice contenant les differents vecteurs d'entrees
%      cible = matrice contenant les vecteurs de sorties attendues
%
% OUT : G0 = population finale de réseaux
%       fitness = vecteur contenant la fitness de chaque individu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function[G0,fitness] = ag(nb_input,nb_output,nb_int,entree,cible)

    % Initialisation du nombre d'individus
    pop = 200;

    % Initialisation des trois poids possibles des synapses
    supp = [-1;0;1];

    % Nombre total de neurones
    N = nb_input + nb_int + nb_output;

    % Initialisation des individus (seuil sur la diag)
    G0 = zeros(N,N,pop);
    for i = 1:pop
        % Matrice de connexions
        G0(:,:,i) = round(1-2*rand(N,N));
        % Seuils sur la diagonale
        for j = 1:N
            G0(j,j,i) = -0.000001;
        end
    end

    % Nombre de generations a effectuer
    nb_gen = 50;

    % Compteur des generations
    k = 1;

    while k < nb_gen

        % Matrices de connexions de la progeniture
        G0_mod = zeros(N,N,pop);
```

```
% Evaluation de la fitness de chaque individu (premiere iteration)
if (k==1)
    fitness = zeros(pop,1);
    for i = 1:pop
        fitness(i) = calcul_erreur(G0(:,:,i),nb_input,nb_output,entree,cible);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%            Crossover               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Sous-intervalle de [0,1] accorde a chaque individu
div = 1/sum(fitness);
nombre = cumsum(fitness*div);

% Taux de crossover
qx = 0.9;

for l = 1:1:(pop/2)
    % Choix des deux individus parents
    ind = zeros(2,1);
    for j = 1:2
        nb = rand(1,1);
        ind(j) = find(nombre>nb,1,'first');
    end
    qy = rand(1,1);
    % Crossover avec proba qx
    if qy <= qx
        % Choix de la colonne a laquelle on coupe la matrice de
        % connexions
        col = ceil(rand(1,1)*(N-1));
        % Echange des colonnes qui suivent
        G0_mod(:,:,((2*l)-1)) = [G0(:,1:col,ind(1)),G0(:,col+1:N,ind(2))];
        G0_mod(:,:,(2*l)) = [G0(:,1:col,ind(2)),G0(:,col+1:N,ind(1))];
    else
        % Elitisme avec proba 1-qx
        G0_mod(:,:,((2*l)-1)) = G0(:,:,ind(1));
        G0_mod(:,:,(2*l)) = G0(:,:,ind(2));
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mutation (dans la population d'enfants) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Taux de mutation
q = 0.09;
```

```
% Nombre total de données
m = pop*N^2;

% Quantite de mutations a effectuer
v = binopdf([0:m],m,q);
[w,z] = max(v);

for i=1:z
    % Choix aleatoire de l'individu a muter
    indiv = ceil(rand(1,1)*pop);
    % Ligne et colonne de l'element modifie
    ligne = ceil(rand(1,1)*N);
    col = ceil(rand(1,1)*N);
    % Si mutation seuil
    if (ligne == col)
        % Empecher mutation pour seuils des noeuds d'entrees
        % (inutile)
        while (ligne <= nb_input)
            ligne = ceil(rand(1,1)*N);
            col = ligne;
        end
        % Variation de la valeur precedente en restant dans [-1,1]
        p = -1 + 2*rand(1,1);
        var = 10000*sign(p)*G0_mod(ligne,col,indiv);
        while (((G0_mod(ligne,col,indiv)+ var) >= 1)||
         ((G0_mod(ligne,col,indiv)+var) <= -1))
            var = 0.1*var;
        end
        G0_mod(ligne,col,indiv) = G0_mod(ligne,col,indiv) + var;
    else
        % Si mutation matrice de connexions, choix d'un autre poids
        % parmi les deux restants
        c = G0_mod(ligne,col,indiv);
        G0_mod(ligne,col,indiv) = randsample(supp(find(supp~=c)),1);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Passage a la generation suivante %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Calcul des fitness des enfants
fitness_mod = zeros(pop,1);
for i = 1:pop
    fitness_mod(i) = calcul_erreur(G0_mod(:,:,i),nb_input,nb_output,entree,cible);
end

% Population totale de '2*pop' individus (G0 U G0_mod)
B = cat(3,G0,G0_mod);
fit = [fitness;fitness_mod];
```

```
        % Rangement des individus par ordre croissant en fonction de leur
        % fitness
        [ordine, indice] = sort(fit);
        temp = B(:,:,indice);
        fit = ordine;
        B = temp;

        % Mise a jour de la generation suivante en gardant les meilleurs
        % individus
        G0(:,:,(pop/10) + 1:pop) = B(:,:,pop + (pop/10) + 1:2*pop);
        fitness((pop/10) + 1:pop) = fit(pop + (pop/10) + 1:2*pop);

        % Insertion d'individus aléatoires
        for i = 1:(pop/10)
            % Matrice de connexions
            G0(:,:,i) = round(1-2*rand(N,N));
            % Seuil sur la diagonale
            for j = 1:nb_input
                G0(j,j,i) = -0.000001;
            end
            for j = nb_input+1:N
                G0(j,j,i) = -1 + 2*rand(1,1);
            end
        end

        % Calcul de la fitness des nouveaux individus
        for i = 1:pop/10
            fitness(i) = calcul_erreur(G0(:,:,i),nb_input,nb_output,entree,cible);
        end

        k = k+1

    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auteur : Nicolay Delphine
% Date : 20/10/2011
% But : calculer la fitness de chaque individu
%
% In : A = matrice de connexions de l'individu
%      nb_input = nombre de neurones d'entrees
%      nb_output = nombre de neurones de sorties
%      entree = matrice contenant les differents vecteurs d'entrees
%      cible = matrice contenant les vecteurs de sorties attendues
%
% Out : fitness = fitness de l'individu represente par la matrice A
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function[fitness] = calcul_erreur(A,nb_input,nb_output,entree,cible)

    % Initialisation de la taille de la matrice de connexions
    n = size(A,1);

    % Separation du vecteur de seuils et de la matrice de connexions
    theta = diag(A);
    W = A - diag(diag(A));

    % Initialisation du nombre d'exemples
    c = size(entree,2);

    % Initialisation de l'erreur sur les sorties et le temps d'execution
    erreur = 0;
    temps = 0;

    % Initialisation de l'erreur maximale pouvant etre commise sur les
    % sorties
    erreur_max = 0;

    % Compteur des exemples
    f = 1;

    % Pour chacun des exemples connus
    while (f <= c)
        % Initialisation du vecteur d'etat des neurones
        x = zeros(n,1);
        % Initialisation des entrees a celle de l'exemple
        x(1:nb_input) = entree(:,f);
        % Initialisation de vecteur des sorties attendues
        target(:,1) = cible(:,f);

        % Temps maximal jusqu'auquel on cherche un cycle
        T = 500;

        % Calcul d'un nombre binaire representant l'etat de x au temps t
        z = 2.^(0:n-1);
        vecteur = zeros(n,T);
        vecteur(:,1) = x;
        valeur = zeros(1,T);
        valeur(1) = z*x;

        t = 2;
        periode = 0;

        % Recherche d'un cycle dans le reseau
        while ((t<T) && (periode == 0))
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            % Calcul du nouvel etat du systeme %
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
            b = W*x;
            % Entrees peuvent pas changer d'ou commence a nb_input+1
            for i = nb_input+1:n
                % Somme des synapses en valeur absolue
                somme = sum(abs(W(i,:)));
                if (somme == 0)
                    % Pas de lien vers le neurone => pas excite
                    x(i) = 0;
                elseif (b(i)/somme >= theta(i))
                    % Excitation superieure au seuil
                    x(i) = 1;
                else
                    % Excitation inferieure au seuil
                    x(i) = 0;
                end
            end

            % Nombre binaire representant le nouvel etat
            vecteur(:,t) = x;
            bin = z*x;
            valeur(t) = bin;

            % Recherche du cycle
            [periode,trans] = trouvecycle(valeur(1:t));
            t = t+1;
        end

        % Si cycle trouve
        if (periode ~= 0)
            % Erreur de sortie
            for i=1:periode
                vect_erreur = abs(vecteur(n-nb_output+1:n,trans+i) - target(:,1));
                erreur = erreur + sum(vect_erreur);
            end
            erreur_max = erreur_max + nb_output*periode;

            % Temps utilise par le reseau
            temps = temps + periode + trans;
        else
            % Pas de cycle trouve => grosse penalite
            erreur = 1;
            erreur_max = 1;
            temps = 1000;
        end

        % Exemple suivant
        f = f+1;

    end
```

```
    % Penalite sur l'ereur commise
    erreur_out = 1 - (erreur/erreur_max);

    % Penalite sur les reseaux plus lents
    temps_max = (2^(n-nb_input))*c;
    erreur_tps = 1 - (temps/temps_max);

    % Penalite sur le nombre de liens du reseau
    somme = 0;
    % somme = nombre de connexions non nulles
    vect_somme = sum(abs(W));
    somme = sum(vect_somme);
    connect_max = n*(n-1);
    erreur_conn = 1 - (somme/connect_max);

    % Erreur totale
    c1 = 0.6;
    c2 = 0.2;
    c3 = 0.2;
    fitness = c1*erreur_out + c2*erreur_tps + c3*erreur_conn;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auteur : Nicolay Delphine
% Date : 20/10/2011
% But : trouver un mouvement periodique dans le comportement du reseau
%       etudie ainsi que le temps de transition avant l'apparition de cette
%       periode
%
% In : vect = vecteur contenant les valeurs a comparer
%
% Out : periode = taille de la periode
%       trans = temps de transition avant l'apparition de la periode
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function[periode,trans] = trouvecycle(vect)

    % Initialisation de val a la derniere valeur du vecteur
    s = size(vect,2);
    val = vect(s);

    % Initialisation de la période et de la periode de transition
    trans = 0;
    periode = 0;

    % Initialisation du compteur et du booleen de la boucle
    trouve = 0;
    i = 1;
```

```
    % Comparaison de chaque element du vecteur avec le dernier tant
    % qu'aucune egalite n'est trouvee
    while ((i<s) && (trouve == 0))
        if vect(i) == val
            % Si decouverte d'une egalite, assignation d'une valeur a la
            % periode et la periode de transition + fin de la recherche
            trouve = 1;
            trans = i-1;
            periode = s-i;
        else
            i = i+1;
        end
    end
end
```

Algorithme génétique multi-objectif NSGA

Cette partie contient les codes qui permettent d'optimiser nos réseaux à l'aide de l'algorithme génétique multi-objectif. Nous présentons ici la fonction réalisant l'algorithme génétique multi-objectif, la fonction qui calcule les valeurs des objectifs et la fonction utilisée pour trier les individus et leur assigner une fitness en fonction de leur rang de non-dominance. Les autres fonctions sont semblables à celles utilisées dans la première partie. Il n'est donc pas utile de les présenter une seconde fois.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auteur : Nicolay Delphine
% Date : 14/03/2012
% But: trouver le meilleur reseau pour modeliser une fonction
%      particuliere
%
% IN : nb_input = nombre d'entrees
%      nb_output = nombre de sorties
%      nb_int = nombre de neurones intermediaires
%      entree = matrice contenant les differents vecteurs d'entrees
%      cible = matrice contenant les vecteurs de sorties attendues
%
% OUT : G0 = population finale de réseaux
%       tab_erreur = matrice contenant les trois objectifs de chaque individu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function[G0,tab_erreur] = ag(nb_input,nb_output,nb_int,entree,cible)

    % Initialisation du nombre d'individus
    pop = 100;

    % Initialisation des trois poids possibles des synapses
    supp = [-1;0;1];

    % Nombre total de neurones
```

```
N = nb_input + nb_int + nb_output;

% Initialisation des individus (seuil sur la diag)
G0 = zeros(N,N,pop);
for i = 1:pop
    % Matrice de connexions
    G0(:,:,i) = round(1-2*rand(N,N));
    % Seuils sur la diagonale
    for j = 1:N
        G0(j,j,i) = -0.000001;
    end
end

% Nombre de generations a effectuer
nb_gen = 10;

% Compteur des generations
k = 1

while k < nb_gen

    % Matrice de connexions de la progeniture
    G0_mod = zeros(N,N,pop);

    % Evaluation des trois objectifs de chaque individu (premiere
    % iteration)
    if (k==1)
        tab_erreur = zeros(pop,3);
        for i = 1:pop
            tab_erreur(i,:) = calcul_erreur(G0(:,:,i),nb_input,nb_output,entree,cible)
        end
        % Calcul de la fitness des individus en fonction de leur rang
        % de non-dominance
        [fitness,vecteur] = fast_nds(N,tab_erreur,G0);
        % Normalisation des fitness
        if (max(fitness) ~= min(fitness))
            fitness = (fitness - min(fitness))/(max(fitness) - min(fitness));
        end
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %             Crossover                 %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Sous-intervalle de [0,1] accorde a chaque individu
    div = 1/sum(fitness);
    nombre = cumsum(fitness*div);

    % Taux de crossover
    qx = 0.8;
```

```
for l = 1:1:(pop/2)
    % Choix des deux individus parents
    ind = zeros(2,1);
    for j = 1:2
        nb = rand(1,1);
        ind(j) = find(nombre>nb,1,'first');
    end
    % Crossover avec proba qx
    qy = rand(1,1);
    if qy <= qx
        % Choix de la colonne a laquelle on coupe la matrice de
        % connexions
        col = ceil(rand(1,1)*(N-1));
        % Echange des colonnes qui suivent
        G0_mod(:,:,((2*l)-1)) = [G0(:,1:col,ind(1)),G0(:,col+1:N,ind(2))];
        G0_mod(:,:,(2*l)) = [G0(:,1:col,ind(2)),G0(:,col+1:N,ind(1))];
    else
        % Elitisme avec proba 1-qx
        G0_mod(:,:,((2*l)-1)) = G0(:,:,ind(1));
        G0_mod(:,:,(2*l)) = G0(:,:,ind(2));
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mutation (dans la population d'enfants) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Taux de mutation
q = 0.09;

% Nombre total de données
m = pop*N^2;

% Quantite de mutation a effectuer
v = binopdf([0:m],m,q);
[w,z] = max(v);

for i=1:z
    % Choix aleatoire de l'individu a muter
    indiv = ceil(rand(1,1)*pop);
    % Ligne et colonne de l'element modifie
    ligne = ceil(rand(1,1)*N);
    col = ceil(rand(1,1)*N);
    % Si mutation seuil
    if (ligne == col)
        % Empecher mutation pour seuils des noeuds d'entrees
        % (inutile)
        while (ligne <= nb_input)
            ligne = ceil(rand(1,1)*N);
```

```
            col = ligne;
        end
        % Variation de la valeur precedente en restant dans [-1,1]
        p = -1 + 2*rand(1,1);
        var = 10000*sign(p)*G0_mod(ligne,col,indiv);
        while (((G0_mod(ligne,col,indiv)+ var) >= 1)||
              ((G0_mod(ligne,col,indiv)+var) <= -1))
            var = 0.1*var;
        end
        G0_mod(ligne,col,indiv) = G0_mod(ligne,col,indiv) + var;
    else
        % Si mutation matrice de connexions, choix d'un autre poids
        % parmi les deux restants
        c = G0_mod(ligne,col,indiv);
        G0_mod(ligne,col,indiv) = randsample(supp(find(supp~=c)),1);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Passage a la generation suivante %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Individus aleatoires a comparer avec les individus de nos
% populations
G0_new = zeros(N,N,(pop/10));
for i = 1:(pop/10)
    % Matrice de connexions
    G0_new(:,:,i) = round(1-2*rand(N,N));
    % Seuils sur la diagonale
    for j = 1:nb_input
        G0_new(j,j,i) = -0.000001;
    end
    for j = nb_input+1:N
        G0_new(j,j,i) = -1 + 2*rand(1,1);
    end
end

% Calcul des fonctions objectifs pour les enfants et les nouveaux
% individus
tab_erreur_mod = zeros(pop,3);
for i = 1:pop
    tab_erreur_mod(i,:) =
    calcul_erreur(G0_mod(:,:,i),nb_input,nb_output,entree,cible);
end

tab_erreur_new = zeros((pop/10),3);
for i = 1:(pop/10)
    tab_erreur_new(i,:) =
    calcul_erreur(G0_new(:,:,i),nb_input,nb_output,entree,cible);
end
```

```
        % Calcul de la fitness des individus en fonction de leur rang de
        % non-dominance. Ce rang est calcule en fonction de tous les
        % individus
        B = cat(3,G0,G0_mod,G0_new);
        tab_r = [tab_erreur;tab_erreur_mod;tab_erreur_new];
        [fit,vecteur] = fast_nds(N,tab_r,B);
        if (max(fit) ~= min(fit))
            fit = (fit - min(fit))/(max(fit) - min(fit));
        end

        % Separation des deux populations et des nouveaux individus
        % introduits
        B = cat(3,G0,G0_mod);
        tab_r = [tab_erreur;tab_erreur_mod];
        fit_new = fit(2*pop+1:2*pop+(pop/10));
        vect_new = vecteur(2*pop+1:2*pop+(pop/10));
        fit = fit(1:2*pop);
        vecteur = vecteur(1:2*pop);

        % Rangement dans l'ordre croissant des individus des deux
        % populations en fonction de la valeur de fit
        [ordine, indice] = sort(fit);
        temp = B(:,:,indice);
        temp2 = tab_r(indice,:);
        temp3 = vecteur(indice);
        fit = ordine;
        B = temp;
        tab_r = temp2;
        vecteur = temp3;

        % Mise a jour de la generation suivante avec les meilleurs
        % individus parmi les deux populations et avec les nouveaux
        % individus
        G0(:,:,((pop/10) + 1):pop) = B(:,:,pop + (pop/10) + 1:2*pop);
        fitness((pop/10) + 1:pop) = fit(pop + (pop/10) + 1:2*pop);
        tab_erreur(((pop/10) + 1):pop,:) = tab_r(pop + (pop/10) + 1:2*pop,:);
        vect((pop/10) + 1:pop) = vecteur(pop + (pop/10) +1:2*pop);

        G0(:,:,1:(pop/10)) = G0_new;
        fitness(1:(pop/10)) = fit_new;
        tab_erreur(1:(pop/10),:) = tab_erreur_new;
        vect(1:(pop/10)) = vect_new;

        k = k+1

    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Auteur : Nicolay Delphine
% Date : 14/03/2012
% But : Calculer la valeur des trois fonctions objectifs pour chaque
%       individu
%
% In : A = matrice de connexions de l'individu
%      nb_input = nombre de neurones d'entrees
%      nb_output = nombre de neurones de sorties
%      entree = matrice contenant les differents vecteurs d'entrees
%      cible = matrice contenant les vecteurs de sorties attendues
%
% Out : erreurs = valeur des trois objectifs pour l'individu represente par
%       la matrice A
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function[erreurs] = calcul_erreur(A,nb_input,nb_output,entree,cible)

    % Initialisation de la taille de la matrice de connexions
    n = size(A,1);

    % Separation du vecteur de seuils et de la matrice de connexions
    theta = diag(A);
    W = A - diag(diag(A));

    % Initialisation du nombre d'exemples
    c = size(entree,2);

    % Initialisation de l'erreur sur les sorties et le temps d'execution
    erreur = 0;
    temps = 0;

    % Initialisation de l'erreur maximale pouvant etre commise sur les
    % sorties
    erreur_max = 0;

    % Compteur des exemples
    f = 1;

    % Pour chacun des exemples connus
    while (f <= c)
        % Initialisation du vecteur d'etat des neurones
        x = zeros(n,1);
        % Initialisation des entrees a celle de l'exemple
        x(1:nb_input) = entree(:,f);
        % initialisation de vecteur des sorties attendues
        target(:,1) = cible(:,f);

        % Temps maximal jusqu'auquel on cherche un cycle
        T = 500;
```

```
% Calcul d'un nombre binaire representant l'etat de x au temps t
z = 2.^(0:n-1);
vecteur = zeros(n,T);
vecteur(:,1) = x;
valeur = zeros(1,T);
valeur(1) = z*x;

t = 2;
periode = 0;

% Recherche d'un cycle dans le reseau
while ((t<T) && (periode == 0))
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Calcul du nouvel etat du systeme %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    b = W*x;
    % Entrees peuvent pas changer d'ou commence a nb_input+1
    for i = nb_input+1:n
        % Somme des synapses en valeur absolue
        somme = sum(abs(W(i,:)));
        if (somme == 0)
            % Pas de lien vers le neurone => pas excite
            x(i) = 0;
        elseif (b(i)/somme >= theta(i))
            % Excitation superieure au seuil
            x(i) = 1;
        else
            % Excitation inferieure au seuil
            x(i) = 0;
        end
    end

    % Nombre binaire representant le nouvel etat
    vecteur(:,t) = x;
    bin = z*x;
    valeur(t) = bin;

    % Recherche du cycle
    [periode,trans] = trouvecycle(valeur(1:t));
    t = t+1;
end

% Si cycle trouve
if (periode ~= 0)
    % Erreur de sortie
    for i=1:periode
        vect_erreur = abs(vecteur(n-nb_output+1:n,trans+i) - target(:,1));
        erreur = erreur + sum(vect_erreur);
    end
    erreur_max = erreur_max + nb_output*periode;
```

```
            % Temps utilise par le reseau
            temps = temps + periode + trans;
        else
            % Pas de cycle trouve => grosse penalite
            erreur = 1;
            erreur_max = 1;
            temps = 1000;
        end

        % Exemple suivant
        f = f+1;

    end

    % Penalite sur l'ereur commise
    erreur_out = 1 - (erreur/erreur_max);

    % Penalité sur les reseaux plus lents
    temps_max = (2^(n-nb_input))*c;
    erreur_tps = 1 - (temps/temps_max);

    % Penalite sur le nombre de liens du reseau
    somme = 0;
    % somme = nombre de connexions non nulles
    vect_somme = sum(abs(W));
    somme = sum(vect_somme);
    connect_max = n*(n-1);
    erreur_conn = 1 - (somme/connect_max);

    % Valeurs des trois objectifs
    erreurs = [erreur_out,erreur_conn,erreur_tps];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auteur: Nicolay Delphine
% Date: 14/03/2012
% But: trier les individus de la population en fonction du critere
%      d'optimalite de Pareto et leur assigner une fitness en fonction de
%      leur rang et de leur proximite avec leurs voisins
%
% IN : N = nombre de neurones des reseaux
%      tab = tableau contenant les valeurs des objectifs pour chaque
%            individu
%      G = matrice en trois dimensions contenant les matrices de connexions
%          de chaque individu
%
% OUT : fitness = vecteur des fitness des individus
%       vecteur = vecteur contenant le rang de chaque individu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function[fitness,vecteur] = fast_nds(N,tab,G)

    % Taille de la population et nombre d'objectifs a optimiser
    pop = size(tab,1);
    M = size(tab,2);

    % Initialisation de la fitness
    fitness = zeros(pop,1);

    % Initialisation de la fitness minimale à pop + eps
    eps = 0.1;
    fmin = pop + eps;

    % Initialisation du compteur de rang et du vecteur contenant le rang de
    % chaque individu
    compt = 1;
    vecteur = zeros(pop,1);

    % Tant que tout les individus n'ont pas ete ranges
    while (norm(tab) > 0)

        % Initialisation du premier individu a regarder
        premier = 1;
        while (norm(tab(premier,:)) == 0)
            premier = premier + 1;
        end
        vecteur(premier) = compt;

        for i = (premier+1):pop
            if (norm(tab(i,:)) > 0)
                % Si l'individu n'est pas deja range, on le met dans le
                % rang en cours
                vecteur(i) = compt;
                j = 1;
                while (j < i)
                    % On compare le nouvel individu avec ceux qui sont deja
                    % dans le rang en cours
                    if (vecteur(j) == compt)
                        l1 = 0;
                        l2 = 0;
                        k1 = 0;
                        k2 = 0;
                        for k = 1:M
                            if (tab(j,k) >= tab(i,k))
                                l1 = l1 + 1;
                            else
                                k2 = k2 + 1;
                            end
                            if (tab(j,k) > tab(i,k))
```

```
                        l2 = l2 + 1;
                    else
                        k1 = k1 + 1;
                    end
                end
                % p domine q si chaque objectif est plus grand ou
                % egal et si au moins un est plus grand
                if ((l1 == M) & (l2 ~= 0))
                    vecteur(i) = 0;
                    % On sort l'element du rang et on arrete la
                    % comparaison
                    j = i;
                % q domine p si chaque objectif est plus grand ou
                % egal et au moins un est plus grand
                else if ((k1 == M) & (k2 ~= 0))
                        % On sort l'element avec lequel on
                        % comparait le nouveau et on continue les
                        % comparaisons pour voir s'il n'est pas
                        % aussi meilleur que les autres elements
                        % deja dans ce rang
                        vecteur(j) = 0;
                    end
                end
            end
            j = j + 1;
        end
    end
end

% On garde les resultats du rang dans un vecteur et on assigne la
% fitness provisoire
niveau = 0;
fitness_prov = 0;
l = 1;
for i = 1:pop
    if (vecteur(i) == compt)
        niveau(l) = i;
        fitness_prov(l) = fmin - eps;
        tab(i,:) = 0;
        l = l+1;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcul de la fitness pour le rang en cours %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Si un seul individu dans la couche --> pas besoin de sharing
if (size(niveau,2) > 1)
```

```
% Nombre d'individus dans la couche
s = size(niveau,2);
% Matrice des distances
d = zeros(s,s);
% Nombre de genes (tous les elements de la matrice)
p = N*N;
% Initialisation de sigma
sigma = 0.5/(10^(1/p));

% On garde les matrices correspondant aux individus de ce rang
% dans un vecteur
GG = zeros(N,N,s);
for l=1:s
    GG(:,:,l) = G(:,:,niveau(l));
end

% Calcul des distances du sharing
for k =1:N
    for l = 1:N
        for i = 1:s
            for j = 1:i-1
                val_min = min(G(k,l,:));
                val_max = max(G(k,l,:));
                if (val_min ~= val_max)
                    d(i,j) =
                    d(i,j) + ((GG(k,l,i) - GG(k,l,j))/(val_min - val_max))^2;
                end
            end
        end
    end
end

for i = 1:s
    for j = s:-1:i+1
        d(i,j) = d(j,i);
    end
end
d = sqrt(d);

% Calcul de la fonction de sharing, du niche count et de la
% fitness reelle
for i=1:s
    m(i) = 0;
    for j = 1:s
        if (d(i,j) <= sigma)
            Sh(i,j) = 1 - (d(i,j)/sigma)^2;
        else
            Sh(i,j) = 0;
        end
        m(i) = m(i) + Sh(i,j);
```

```
                end
                fitness_prov(i) = fitness_prov(i)/m(i);
            end

            for i = 1:s
                % Mise a jour des fitness
                fitness(niveau(i)) = fitness_prov(i);
            end

            % Mise a jour de la finess min (les individus du rang suivant
            % ont des fitness strictement inferieures a celle du rang en
            % cours)
            fmin = min(fitness_prov);

        else
            % S'il ne reste qu'un individu dans le rang --> pas de sharing
            fitness(niveau) = fitness_prov;
        end

        compt = compt + 1;
    end
end
```