Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Moteur d'intelligence artificielle pour jeux de gestion multijoueurs

Calay, Fabrice; Chenal, Bertrand

Award date:
2006

Link to publication

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

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

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

Download date: 02. May. 2026

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

Année académique 2005-2006

Moteur d'intelligence artificielle pour jeux de gestion multijoueurs

CALAY Fabrice CHENAL Bertrand

Mémoire présenté pour l'obtention du grade de
Licencié en informatique

Nous tenons à remercier notre promoteur Jean-Paul Leclercq pour son aide précieuse et son soutien tout au long de la réalisation de ce mémoire. Merci également à Jean-Marie Jacquet et Samuel Marin pour leur aide dans la récolte de documents utiles à la rédaction de ce travail.

Résumé

Ce travail a pour but de développer un moteur d'intelligence artificielle pour des jeux de gestion multijoueurs. Plusieurs techniques bien connues dans le domaine de l'intelligence artificielle sont étudiées. Une analyse critique de celles-ci est ensuite effectuée afin de déterminer celles qui s'adaptent au mieux à cette classe de jeux. Nous terminons sur une étude de cas permettant d'appuyer notre analyse.

Mots-clés : Intelligence artificielle, jeux de gestion, réseaux de neurones, algorithmes évolutionnaires, apprentissage automatique.

Abstract

The aim of this work is to develop an artificial intelligence for multiplayer management games. First several well known techniques in artificial intelligence field are discussed. Critical analysis of these ones is set to determine which fit the best to our class of games. A study case that allow to strengthen our analysis conclude this work.

Keywords : Artificial intelligence, management games, neural networks, evolutionnary programming, machine learning.

Table des matières

Introduction **3**

1 Techniques d'intelligence artificielle **5**

1.1 Arbres de recherche 6
1.1.1 Méthode Depth-First 7
1.1.2 Méthode Breadth-First 9
1.1.3 Principe minimax 10
1.2 Réseaux neuronaux 14
1.2.1 Introduction aux réseaux de neurones 14
1.2.2 Présentation historique 15
1.2.3 Structure d'un neurone artificiel 17
1.2.4 Architecture d'un réseau 21
1.3 Algorithmes évolutionnaires 25
1.3.1 Principes de base 27
1.3.2 Eléments théoriques 30
1.3.3 Extensions 35

2 Apprentissage automatique **37**

2.1 Induction - Déduction 38
2.2 Réseaux de neurones 46
2.2.1 Apprentissage supervisé 46
2.2.2 Apprentissage non supervisé 51

3 Etat de l'art **55**

3.1 Application à la robotique 55
3.2 Application aux jeux vidéos 60
3.3 Application à l'optimisation mathématique 62
3.4 Prévisions cinématographiques 62
3.5 Prévisions boursières 66

3.6 Discussion 68

4 Etude de cas **69**

4.1 Présentation du jeu 70

4.1.1 Contenu du jeu 70

4.1.2 Mise en place 70

4.1.3 Déroulement du jeu 71

4.1.4 Piocher, jouer ou défausser 71

4.1.5 Cartes 71

4.1.6 S'emparer des cartes Galion 71

4.1.7 Fin de la partie & Décompte des points 72

4.2 Intelligence artificielle 73

4.2.1 Algorithmes évolutionnaires 73

4.2.2 Réseaux neuronaux 80

4.2.3 Discussion 86

4.3 Implémentation 87

Conclusion **93**

Bibliographie **94**

Annexes **98**

A Identifiants des cartes **101**

A.1 Cartes Galion 101

A.2 Cartes Pirates 101

A.2.1 Couleur Rouge 101

A.2.2 Couleur Bleu 102

A.2.3 Couleur Vert 102

A.2.4 Couleur Jaune 102

B Documents disponibles **103**

Introduction

A ses débuts, l'idée de permettre à un ordinateur de raisonner comme un être humain semblait utopique. Cependant, suite aux avancées technologiques et à l'arrivée de nouveaux concepts théoriques, les machines n'ont cessé d'évoluer afin de permettre à l'homme de se libérer de certaines tâches. Par la suite, l'industrie s'est intéressée au développement de ces différentes techniques et est parvenue à exploiter ces résultats dans le cadre de problèmes professionnels.

Ces progrès technologiques ont également profité aux jeux informatisés qui se sont enrichis au fil des années. Cette complexification s'accompagne de la nécessité de développer des intelligences artificielles toujours plus abouties. C'est ce type de moteur que nous avons entrepris d'établir pour un jeu de gestion multijoueurs.

Dans un premier chapitre, nous présentons des techniques d'intelligence artificielle pouvant être apparentées aux programmes de jeux. Ensuite, nous montrons la manière dont un ordinateur peut apprendre automatiquement afin de répondre au mieux aux attentes de l'utilisateur. Dans un troisième chapitre, nous exposons différentes applications des techniques présentées. Suite à une analyse critique de celles-ci, nous entreprenons une implémentation d'un moteur d'intelligence artificielle pour un jeu de gestion multijoueurs. Cette étude expérimentale nous permet de dégager les principales difficultés de la conception d'un tel moteur.

Chapitre 1

Techniques d'intelligence artificielle

Au cours de l'évolution des ordinateurs, l'homme n'a cessé de vouloir rendre ceux-ci assez intelligents pour l'aider dans sa prise de décisions. Dans ce chapitre, nous nous concentrons sur les techniques d'intelligence artificielle utilisées dans le cadre d'un jeu.

Nous commençons par une introduction basée sur les arbres de recherche afin d'avoir une idée du mécanisme utilisé dans le cadre de jeux à deux joueurs, où le nombre de coups est relativement limité. Ensuite, nous présentons deux techniques plus élaborées que sont les réseaux de neurones et les algorithmes évolutionnaires, celles-ci étant plus fréquemment utilisées dans les jeux multijoueurs (plus de deux).

1.1 Arbres de recherche

Dans les jeux à deux joueurs, la plupart des algorithmes utilisés pour représenter une intelligence artificielle utilise un arbre de recherche. En d'autres termes, les possibilités de jeu offertes à un joueur au cours de son tour de jeu sont représentées par un nœud de l'arbre. L'algorithme utilisé par la machine examine cet arbre afin de choisir un chemin dans celui-ci qui emmènera le joueur électronique à une victoire (équivalente à une défaite de l'autre joueur). La figure 1.1 représente un de ces arbres de recherches pour le jeu *Tic-Tac-Toe.*

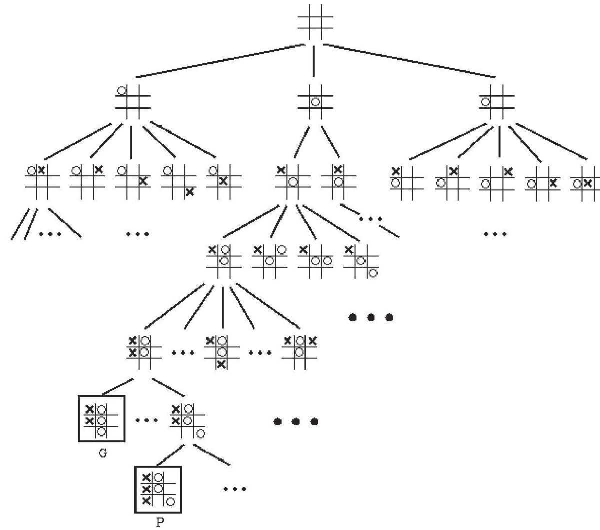

FIG. 1.1 – *Arbre de recherche pour le jeu Tic-Tac-Toe [55].*

Cette méthode est efficace lorsque la taille de l'arbre ne prend pas des proportions immenses. Lors d'un jeu d'échec, celui-ci peut prendre une dimension considérable et il est nécessaire de disposer d'une machine d'une grande puissance et d'utiliser différentes extensions (élagage, bibliothèque d'ouvertures, ...) pour obtenir des résultats en des temps raisonnables. Pendant de longues années, de nombreux informaticiens se sont penchés sur le sujet afin d'obtenir une machine assez puissante et un algorithme assez performant pour vaincre un joueur d'échec. Ce n'est qu'en 1997 que *Deep Blue* obtenu une victoire par deux manches à une face à Gary Kasparov, le meilleur joueur du monde à cette époque.

1.1.1 Méthode Depth-First

La méthode Depth-First (ou méthode en profondeur d'abord) explore les nœuds de l'arbre de recherche de la façon suivante. Pour un nœud particulier, nous choississons une règle de production n'ayant pas encore été utilisée. Si le nœud s'avère être une feuille de l'arbre, nous remontons au nœud parent et choisissons une autre règle de production parmi celles n'ayant pas encore été utilisées.

Cette méthode est fréquemment utilisée, cependant, elle présente un inconvénient. En effet, elle nécessite parfois de parcourir entièrement l'arbre de recherche avant de parvenir à la solution souhaitée. De plus, si celui-ci contient des cycles et que l'algorithme établi ne marque pas les sommets déjà visités, nous obtenons des branches infinies qui entrainent un bouclage interminable lors de la recherche.

L'avantage de cette méthode réside dans le fait qu'elle ne nécessite qu'un espace mémoire réduit car toute l'exploration peut se faire dans un espace de stockage d'une branche de taille n (n étant le nombre de niveaux de l'arbre de recherche).

Cette méthode est essentiellement appliquée dans des problèmes à espaces d'états réduits afin d'éviter ce type de problèmes. Cependant, il s'agit de la méthode utilisée par le langage *PROLOG*. C'est pourquoi il est essentiel de prendre garde à l'ordre donné aux différentes clauses d'un programme *PROLOG* lors de sa rédaction.

L'algorithme de recherche en profondeur d'abord peut s'énoncer de la manière suivante :

Algorithme Depth-First :

Entrée : Graphe $G = (N,A)$, Sommet s

M : structure de données ordonnée LIFO LastIn/FirstOut (dernier entré/premier sorti)
Initialiser tous les sommets à non-marqué; Marquer s
$M \leftarrow s$
TantQue M n'est pas vide
    $x \leftarrow M$ où $x = first(M)$
    Pour chaque fils y non-marqué de x
        Marquer y
        $M \leftarrow y$
    FinPour
FinTantQue

1.1.2 Méthode Breadth-First

La méthode en largeur d'abord (Breadth-First) explore l'arbre de recherche niveau par niveau. Cette technique a besoin de ressources mémoires plus importantes que la méthode en profondeur d'abord car elle nécessite le stockage des différents niveaux. De plus, elle peut également être amenée à parcourir l'entièreté de l'arbre de recherche avant de parvenir à une solution.

Son principal avantage réside dans le fait qu'elle permet de trouver une solution moins profonde, c'est-à-dire, par exemple, dans le cadre d'un jeu, qu'elle permet d'obtenir des résultats similaires à une solution située à un niveau supérieur en un nombre inférieur de coups.

L'algorithme de recherche en largeur d'abord peut s'énoncer de la manière suivante :

Algorithme Breadth-First :

Entrée : Graphe $G = (N,A)$, Sommet s

M : structure de données ordonnée FIFO FirstIn/FirstOut (premier entré/premier sorti)
Initialiser tous les sommets à non-marqué; Marquer s
$M \leftarrow s$
TantQue M n'est pas vide
    $x \leftarrow M$ où $x = first(M)$
    Pour chaque fils y non-marqué de x
        Marquer y
        $M \leftarrow y$
    FinPour
FinTantQue

1.1.3 Principe minimax

Afin d'obtenir un joueur intelligent, il est nécessaire d'ajouter une fonction d'évaluation à un algorithme de recherche. En effet, celle-ci permet de cibler de manière plus pointue les différentes branches de l'arbre qui pourraient s'avérer meilleures que les autres.

Dans le cas de programmes de jeux à deux joueurs, les algorithmes utilisés par chacun d'eux sont antagonistes. Ainsi, à partir d'une position de départ, l'ordinateur génère l'ensemble des positions atteignables. Ensuite, à partir chacune de ces positions, l'ensemble des positions atteignables par l'adversaire est généré.

Cet arbre généré peut prendre des proportions importantes et il est donc nécessaire de se limiter dans le choix des positions générées. En effet, même le plus puissant ordinateur au monde ne pourrait générer l'ensemble des positions de jeu aux échecs en un temps raisonnable.

Ensuite, lorsque cet arbre est généré, une valeur est attribuée à chaque feuille par l'intermédiaire d'une fonction d'évaluation. L'ordinateur choisit ensuite la position à atteindre à partir du niveau 0 de l'arbre pour obtenir la position finale (feuille) ayant la meilleure estimation. Il s'agit donc de *jouer le coup* garantissant le gain maximal suite à une défense quelconque de l'adversaire. Cette méthode suppose que l'adversaire utilise également le même type de méthodes. Ce mécanisme est appelé *principe minimax*.

Etant donné que ce type d'arbres de recherche est établi pour des jeux à deux joueurs, les niveaux de celui-ci ont une signification différente. En effet, d'une part, un niveau peut représenter une étape maximisante, c'est-à-dire un niveau où le joueur désire d'augmenter ses chances de victoire. D'autre part, il peut également représenter une étape minimisante, c'est-à-dire que le joueur prévoit que son adversaire joue son *meilleur coup*. L'arbre de recherche est donc composé en alternance de ces deux types de niveaux.

Algorithme minimax

Le principe de l'algorithme *minimax* est d'obtenir une estimation pour chacun des nœuds de l'arbre de recherche. En effet, la fonction d'évaluation n'est applicable qu'aux nœuds terminaux et il est donc nécessaire de définir une méthode pour donner une valeur aux nœuds des niveaux inférieurs. Cet algorithme procède *en profondeur d'abord*. En effet, une branche complète (de profondeur n) de l'arbre de recherche est générée. Ensuite, le nœud situé au niveau $n-1$ dans celle-ci est marqué par la valeur obtenue par la fonction d'évaluation appliquée au nœud terminal. Ensuite, la feuille suivante est générée et son estimation est utilisée pour éventuellement mettre à jour la valeur du nœud parent. Cette mise à jour dépend de la nature du *coup à jouer*. Après avoir exploré toutes les feuilles atteignables à partir de ce nœud, le nœud situé au niveau $n-2$ est marqué avec la valeur du nœud du niveau $n-1$ (toujours dans la branche de taille n en mémoire) et l'opération est répétée jusqu'à ce que toutes les feuilles et tous les niveaux aient été générés.

Algorithme α - β

L'algorithme *minimax* présente un inconvénient car il nécessite l'examination de l'ensemble des nœuds de l'arbre de recherche. En effet, suite au principe de *minimax*, un certain nombre de nœuds n'ont pas besoin d'être explorés. C'est le principe adopté par l'algorithme α - β.

Nous allons illustrer cette méthode par un exemple. La figure 1.2 présente un arbre de recherche ayant une étape minimisante et une étape maximisante. Comme nous pouvons le constater, suite à l'estimation du deuxième nœud du premier niveau, nous pouvons conclure que la racine aura au moins une valeur égale à 3 car elle se trouve sur une étape maximisante. L'analyse du troisième nœud nous amène aux conclusions suivantes. Etant donné que la première feuille lié à ce nœud est évalué à 1 et que celui-ci se trouve sur une étape minimisante, nous pouvons conclure que le troisième nœud aura une valeur inférieure ou égale à 1. Par conséquent, cette branche de l'arbre sera inutilisé par la suite et l'évaluation des feuilles restantes s'avère sans importance.

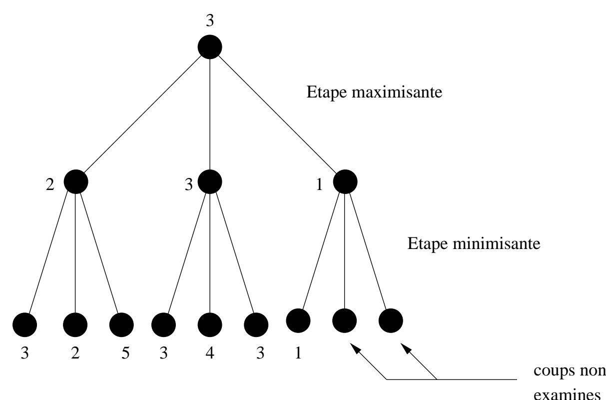

FIG. 1.2 – *Parcours α - β d'un arbre de jeu [3].*

Ainsi, en résumé, lorsqu'un nœud atteint un certain seuil, il s'avère inutile d'explorer la descendance encore non-examinée de ce nœud. Dans l'exemple que nous venons de présenter, certaines feuilles de l'arbre de recherche ne sont pas examinées. Cependant, l'élagage α - β peut concerner d'autres niveaux que les feuilles terminales et ainsi permettre de négliger totalement une grande partie de l'arbre de recherche. Cette opération peut donc mener à de nettes améliorations lors de la recherche d'une solution.

Deux seuils sont établis au cours de l'algorithme : α pour les nœuds de type *Min* et β pour les nœuds de type *Max* :

- Le seuil α : pour un nœud de type *Min* n, il s'agit de la plus grande valeur (connue) de tous les nœuds de type *Max* ancêtres de n. Si n atteint une valeur inférieure à α, il s'avère inutile d'explorer sa descendance.

- Le seuil β : pour un nœud de type *Max* n, il s'agit de la plus petite valeur (connue) de tous les nœuds de type *Min* ancêtres de n. Si n atteint une valeur supérieure à β, il s'avère inutile d'explorer sa descendance.

Ces deux seuils sont mémorisés durant le parcours de l'arbre en profondeur durant l'exécution de l'algorithme. Afin d'effectuer l'algorithme α - β sur un arbre de recherche, la procédure citée ci-dessous [3] sera appelée avec les paramètres $n = Racine$, $\alpha = -\infty$ et $\beta = +\infty$.

Procédure α - β (n, α, β) :

Si n est terminal,
  alors Retourner $h(n)$
Sinon Si n est de type *Max*
  Soit $(f_1 \dots f_k)$ les fils de n
  Pour j alllant de 1 à k et **tant que** $\alpha < \beta$ **faire**
    $\alpha \leftarrow \max(\alpha, \alpha\text{ - }\beta\ (f_j, \alpha, \beta))$
  Fin-faire
  Retourner α
Sinon Si n est de type *Min*
  Soit $(f_1 \dots f_k)$ les fils de n
  Pour j alllant de 1 à k et **tant que** $\alpha < \beta$ **faire**
    $\beta \leftarrow \min(\beta, \alpha\text{ - }\beta\ (f_j, \alpha, \beta))$
  Fin-faire
  Retourner β
Fin-Si

1.2 Réseaux neuronaux

1.2.1 Introduction aux réseaux de neurones

Les réseaux de neurones artificiels (ou RNA) constituent une technique de traitement de l'information inspirée par le fonctionnement des systèmes nerveux biologiques. Ils sont composés d'un grand nombre d'éléments (les neurones) interconnectés, chacun d'entre eux réalisant une petite partie du traitement. Les réseaux de neurones artificiels sont capables de résoudre des problèmes spécifiques via un apprentissage par l'exemple. Quand celui-ci est terminé, le réseau est capable de résoudre la classe de problèmes pour laquelle il a été entraîné, et ce même avec un apprentissage restreint. Nous espérons qu'un *bon* réseau de neurones soit être capable de faire de l'induction sur l'ensemble de la classe de problèmes à partir d'un nombre d'exemples limités.

Aux cours des dernières années, les réseaux neuronaux ont fait l'objet de nombreuses études et applications . Ces dernières interviennent dans des domaines les plus variés tels que la reconnaissance d'images, l'analyse de symptômes médicaux, la prédiction des cours de la bourse ou l'anticipation des pannes de moteurs. En pratique, les réseaux de neurones peuvent être appliqués dans les situations pour lesquelles il existe une relation entre les variables en entrée et les variables de sortie et ce même si cette relation est complexe ou cachée. Ils sont particulièrement indiqués pour les problèmes contenant un très grand nombre de variables ou pour ceux dont les données sont difficilement interprétables. Enfin, l'utilisation des RNA représente un ultime recours quand aucun algorithme connu ne solutionne une question donnée.

Le concept de réseau de neurones artificiels est en soit intellectuellement séduisant puisqu'il se base sur une représentation simplifiée du cerveau humain. De plus, d'autres sources d'intérêt justifient l'utilisation des RNA. D'abord, ils sont très puissants, dans le sens où ils peuvent modéliser des situations très complexes. Ensuite, ils sont faciles à utiliser puisqu'ils sont capables d'apprendre par eux-mêmes, de façon supervisée ou non.

Néanmoins la médaille possède un revers. Une fois calibrés, les réseaux de neurones sont similaires à des boîtes noires et sont difficilement interprétables. Ils permettent de résoudre des problèmes mais ne nous font pas avancer sur l'étude du problème lui-même.

1.2.2 Présentation historique

Les réseaux neuronaux tels que nous les dénommons ici sont bien entendu des réseaux artificiels. Ils sont conceptuellement inspirés par le système nerveux des organismes vivants : Ceux-ci sont constitués de neurones capables de transporter des influx nerveux et sont connectés par des synapses qui assurent la propagation du signal. Un neurone est composé notamment de dendrites et d'un axone qui sont respectivement responsables de la détection de l'influx et de sa propagation. Tous ces différents éléments constituant le système nerveux humain sont représentés à la figure 1.3.

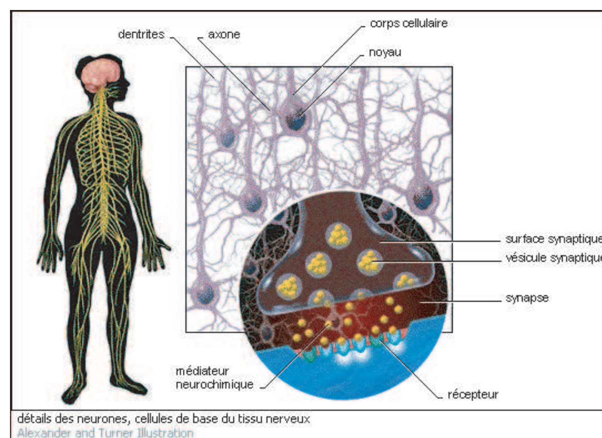

FIG. 1.3 – *Représentation d'un neurone humain [41].*

Pour terminer notre passage en revue des principaux concepts qui sont à l'origine des réseaux de neurones artificiels, remarquons que le cerveau humain, qui est le siège de la mémoire et des raisonnements, constitue un réseau de très nombreux neurones fortement interconnectés.

Les principes de bases d'un système nerveux ont permis dès 1943 au neurophysiologiste Warren McCulloch et au logicien Walter Pitts de reproduire

des portes logiques telles que les portes ET ou OU, et ce à partir d'un modèle standard[1] de neurone. A l'époque, la communauté scientifique cherchait à savoir si le cerveau pouvait être simulé par une machine de Turing[2]. C'est dans ce contexte que McCulloch et Pitts ont montré qu'un réseau de neurones est l'équivalent d'une machine de Turing universelle[3].

Par la suite, en 1959, Bernard Widrow et Marcian Hoff, de l'université de Standford, ont conçu deux modèles, appelés ADALINE et MADALINE (Multiple ADAptative LINear Elements). ADALINE a été développé afin de prédire, à partir d'un pattern de bit, le bit suivant dans un flux de données. Le modèle MADALINE a été le premier réseau de neurones à être appliqué à un problème industriel. Il s'agissait de créer un filtre éliminant les échos sur les lignes téléphoniques.

1. dit de McCulloch - Pitts
2. Une machine de Turing est une machine capable de simuler la logique de n'importe quel ordinateur pouvant être construit [40].
3. Une machine de Turing universelle est une machine de Turing capable de simuler toute autre machine de Turing [40].

1.2.3 Structure d'un neurone artificiel

Un neurone artificiel est un élément avec plusieurs entrées et une seule sortie (cfr Figure 1.4). Cette sortie est gouvernée par deux facteurs : les valeurs en entrée et une règle. Celle-ci,appelée aussi fonction d'activation, détermine la sortie à partir des valeurs en entrée.

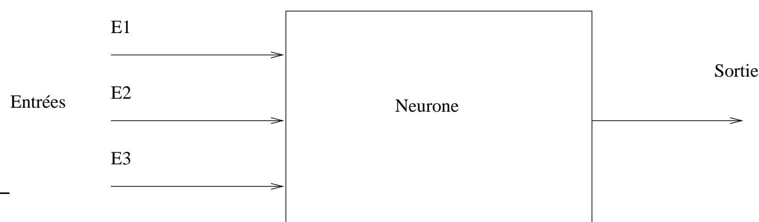

FIG. 1.4 – *Structure générale d'un neurone.*

Premier exemple de neurone artificiel

Considérons que les valeurs en entrées possèdent deux états : *actif* (zéro) ou *inactif* (un) et que la règle d'activation dépend de la configuration de l'ensemble de ces valeurs.

Pour entraîner ce neurone, plusieurs configurations sont données en entrée. Pour chacune d'elles, une valeur de sortie désirée est fournie. Ainsi, le neurone possède une liste de configurations possibles et pour chacun de ces éléments il sait s'il doit s'activer (fournir *un* en sortie) ou pas.

Une fois cette phase d'entrainement terminée, le neurone possède deux ensembles. Un ensemble de configurations pour lesquelles le neurone doit s'activer (ensemble positif) et un ensemble de configurations pour lesquelles il ne doit pas s'activer (ensemble négatif). Ces ensembles ne recouvrent pas de manière exhaustive toutes les configurations possibles. Dans le cas où le neurone reçoit en entrée une configuration pour laquelle il n'a pas été entrainé, celui-ci va appliquer une distance de Hamming :

Définition 1.1 (Distance de Hamming) *La distance de Hamming entre deux séquences binaires de même taille est égale au nombre de bits de rang identique qui diffèrent. Par exemple :* $d(1100,1010) = 2$

La fonction d'activation va donc générer une valeur de sortie si la configuration en entrée est plus proche (au sens de Hamming) de l'ensemble positif que de l'ensemble négatif.

Neurone de McCulloch et Pitts

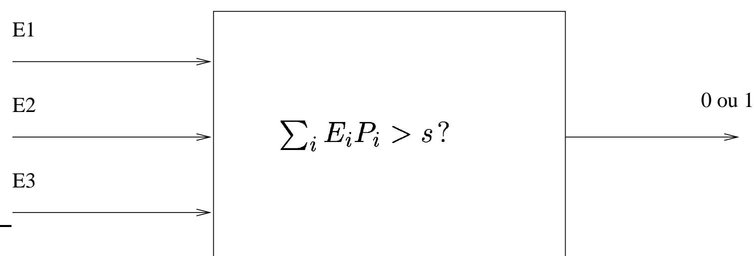

FIG. 1.5 – *Neurone de McCulloch et Pitts.*

La figure 1.5 représente un neurone dit *de McCulloch et Pitts*. Celui-ci possède en entrée des valeurs qui sont pondérées, ce qui permet de définir une **fonction d'entrée** :

$$\sum_i x_i p_i$$

où

$$x_i = \begin{cases} 1 & \text{si l'entrée } i \text{ est active,} \\ 0 & \text{sinon} \end{cases}$$

et p_i est le poids correspondant au $i$ème arc incident au neurone. La quantité s reprise à la Figure 1.5 est une valeur *seuil*, choisie de sorte que le neurone génère une valeur de sortie une fois ce seuil dépassé, c'est-à-dire lorsque

$$\sum_i x_i p_i > s.$$

Ce type de neurones peut s'adapter à de nombreuses situations en jouant sur les poids et sur le seuil. A titre d'exemple, considérons un neurone possédant deux entrées. Si nous choisissons des poids $P_1 = P_2 = 1$ et un seuil $s = 1$, et si nous interprétons une entrée égale à 1 comme vrai, ce neurone implémente une porte ET. Quant à un seuil $s = 0$, il permettra d'obtenir une porte OU. Pour définir une implication, nous choisissons $P_1 = -1$, $P_2 = 1$ et $s = -1$.

Version probabiliste

Un neurone probabiliste permet de générer une valeur de sortie en fonction des valeurs d'entrée avec une certaine probabilité. Dès lors son comportement se rapproche de celui d'un organisme vivant en le rendant moins déterministe.

Dans un neurone de ce type, le seuil n'est pas fixé mais est choisi grâce à une fonction aléatoire. Cette fonction respecte des lois probabilistes telles qu'une loi normale (en forme de cloche de Gauss) ou une loi logistique (ou distribution de Boltzmann) comme illustrées à la figure 1.6.

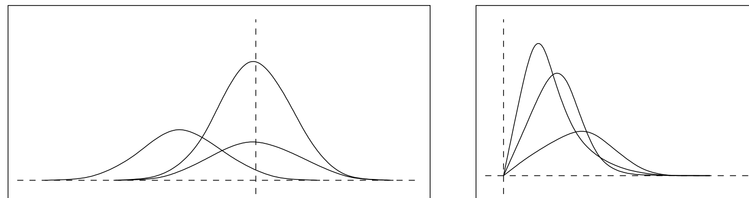

FIG. 1.6 – *Courbes de Gauss et Boltzmann.*

La fonction d'activation du neurone se définit comme suit :

$$f(x) = \begin{cases} 0 & \text{si } \sum_i x_i p_i \leq s; \\ 1 & \text{si } \sum_i x_i p_i > s. \end{cases}$$

où s est un seuil appartenant à $[-\infty,\infty]$ et déterminé aléatoirement en suivant la loi choisie.

Généralisation

La généralisation du neurone de McCulloch et Pitts consiste à garder la même fonction d'entrée ou une généralisation de celle-ci comme un produit, un maximum ou un minimum et de lui appliquer une fonction d'activation qui définit la valeur de sortie (cfr Figure 1.7).

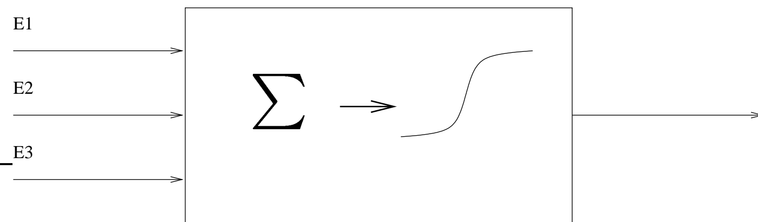

FIG. 1.7 – *Généralisation du neurone de McCulloch et Pitts.*

La fonction d'activation peut être entre autres une fonction linéaire, une fonction de Heaviside, ou une sigmoïde (cfr Figure 1.8).

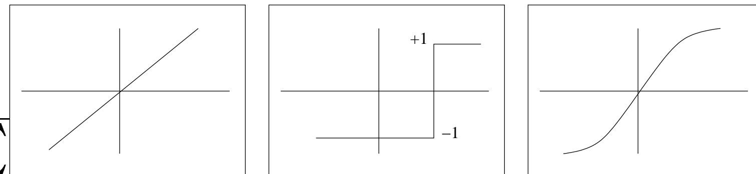

FIG. 1.8 – *Exemples de fonctions d'activation.*

Le type de neurones utilisé par Widrow et Hoff pour mettre en place ADALINE est en fait un neurone de ce type où la fonction de sortie est une simple fonction linéaire.

1.2.4 Architecture d'un réseau

A partir d'opérations simples réalisées par la brique de base qu'est le neurone, un agencement structuré permet d'accomplir des tâches plus complexes. Un réseau de neurones interconnecte les sorties de certains neurones avec les entrées d'autres neurones. Il existe 2 types d'architectures différentes : les réseaux *feed-forward* et les réseaux *feed-back*.

Les réseaux feed-forward

Pour ce type d'architectures, le signal parcourant le réseau ne se propage que dans un seul sens, depuis les neurones d'entrées vers les neurones de sorties. Le réseau ne possède donc pas de boucle. En terme de graphes, un tel réseau forme une arborescence : c'est-à-dire un graphe orienté acyclique connexe où chaque sommet est de degré entrant au plus un. Nous présentons deux modèles classiques de réseaux *feed-formard*.

Le premier modèle est appelé perceptron[4] à simple couche[5]. Il s'agit du type le plus simple de réseaux de neurones. Il est constitué d'une couche de neurones directement reliée aux valeurs d'entrée du problème. A chaque couple entrée - neurone est associé un poids. La figure 1.9 présente un exemple de perceptron simple couche.

4. Le terme *perceptron* désigne généralement des réseaux dit "feed-forward" mais pour certains auteurs il désigne un cas particulier de réseau "feed-forward" où la couche d'input subit un pré-traitement.

5. Le terme *simple couche* désigne un réseau de neurones ne possédant qu'une couche d'entrée et une couche de sortie.

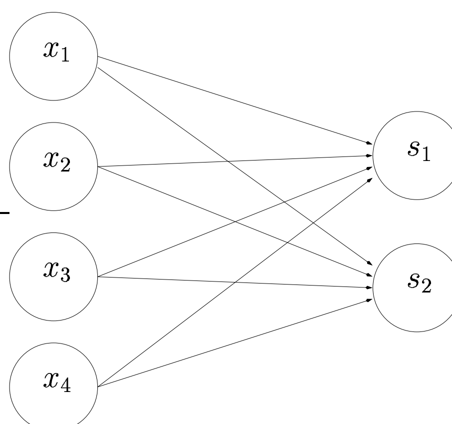

FIG. 1.9 – *Exemple de perceptron simple couche.*

Le second modèle est celui du perceptron multi-couches. Similaire au réseau à simple couche, il possède des couches intermédiaires (qui sont appelées couches cachées) qui permettent de réaliser des traitements plus complexes. Il va de soi qu'un réseau multi-couches n'a de sens que si la fonction d'activation est non-linéaire. La figure 1.10 présente un exemple de réseaux de neurones multi-couches.

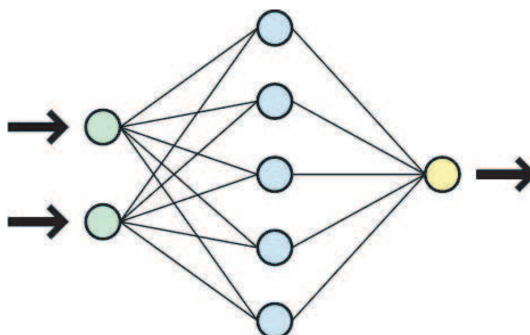

FIG. 1.10 – *Exemple de perceptron multi-couches.*

En 1969, Marvin Minsky et Seymour Papert ont montré que les perceptrons à simple couche sont incapables d'implémenter une porte XOR (c'est-à-dire un OU exclusif). Ils en ont conclu mais de manière trop hâtive que les perceptrons multi-couches présentaient les mêmes lacunes. Malheureusement, ces résultats ont ralenti les travaux de recherche dans le domaine. Les

effort des chercheurs ne fûrent de retour qu'une quinzaine d'années plus tard à la suite d'autres travaux démontrant les possibilités offertes par les réseaux multi-couches.

Les réseaux feed-back

Ce type d'architecture est caractérisé par la présence de boucles dans le réseau (appellée dans ce cas réseau récursif). Plus puissante, cette structure permet de créer des réseaux plus complexes. De plus, ces boucles introduisent une dynamique puisqu'elles peuvent entrainer plusieurs changements d'état (ou plus généralement de valeur de sortie) des mêmes neurones. Dans de tels réseaux, le signal stoppe sa propagation quand un équilibre est atteint.

Les réseaux récursifs peuvent soit être des réseaux *feed-forward* dans lequel des boucles ont été introduites, soit des réseaux complètement récursifs. Dans ce cas, il n'existe pas de structure en couches, chaque neurone reçoit des données en provenance de tous les autres neurones. En général, seul un sous-ensemble de neurones reçoit des données en entrée et un autre sous-ensemble génère une sortie. La figure 1.11 présente un ensemble de neurones totalement interconnectés, celui-ci pouvant constituer une partie d'un réseau de neurones de taille plus importante.

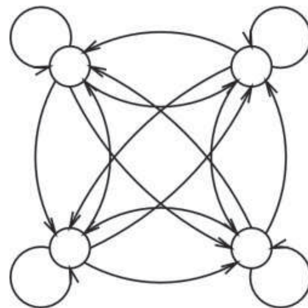

FIG. 1.11 – *Exemple de réseau récursif.*

Les réseaux de Hopfield constituent un cas particulier des réseaux récursifs pour lesquels toutes les connexions sont symétriques[6]. Cette contrainte, caractéristique des réseaux de Hopfield, permet de garantir que la convergence de la dynamique. En effet, en raison de la récursivité du réseau, chaque neurone peut influencer les neurones responsables de son excitation, ce qui modifie la sortie de ceux-ci et influence en cascade d'autres neurones. En pratique, les réseaux de Hopfield peuvent être utilisés pour établir une mémoire associative (*adressable par le contenu*). Par exemple, un tel réseau est capable d'associer une image déteriorée à l'image originale si celle-ci lui a été soumise auparavant.

6. Une connexion a-b de poids p est dite symétrique s'il existe une connexion b-a de poids p.

1.3 Algorithmes évolutionnaires

Comme nous venons de le voir dans la section précédente, la technique des réseaux de neurones a été dérivée du réseau de neurones au sens de la médecine. C'est n'est pas la seule technique de programmation que les mécanismes naturels ont inspiré à l'informatique. C'est le cas notamment des algorithmes évolutionnaires. Ceux-ci se basent sur la théorie de l'évolution de Darwin, seuls les êtres les mieux adaptés au milieu survivent et se reproduisent.

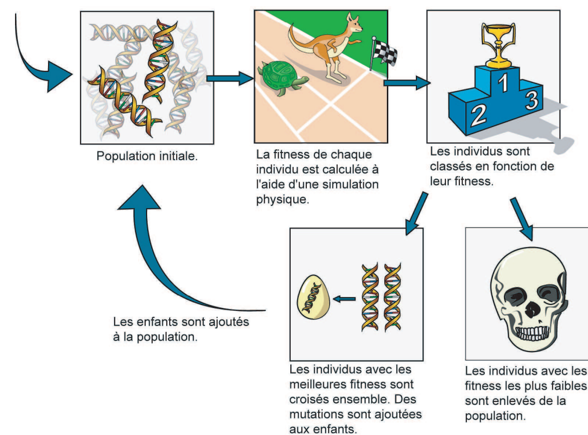

FIG. 1.12 – *Principe d'un algorithme évolutionnaire [7].*

Les algorithmes évolutionnaires forment une classe d'algorithmes itératifs basés sur cette évolution. Au départ d'un tel algorithme, nous disposons d'une population initiale que nous pouvons évaluer à l'aide d'une fonction de performance. Les meilleurs candidats seront retenus pour former la population suivante et ainsi de suite jusqu'à l'obtention d'une solution optimale au problème. Lors des différentes itérations de l'algorithme, plusieurs opérations peuvent être effectuées afin de converger plus rapidement vers une solution ou de parcourir de manière plus performante l'espace de recherche. La figure 1.12 illustre cette méthode.

Lors d'une itération d'un algorithme évolutionnaire, les opérations suivantes peuvent être effectuées :

- Sélection des candidats allant se reproduire,
- Recombinaison des candidats,
- Mutation des candidats,
- Sélection des survivants pour la génération suivante.

Plusieurs méthodologies existent au sein des algorithmes évolutionnaires :

- Evolutionary Programming (EP)
- Evolutionary Strategies (ES)
- Genetic Algorithms (GA)

Les opérations citées précédemment ne sont pas nécessairement effectuées par toutes les méthodologies. La figure 1.13 présente ces différents choix.

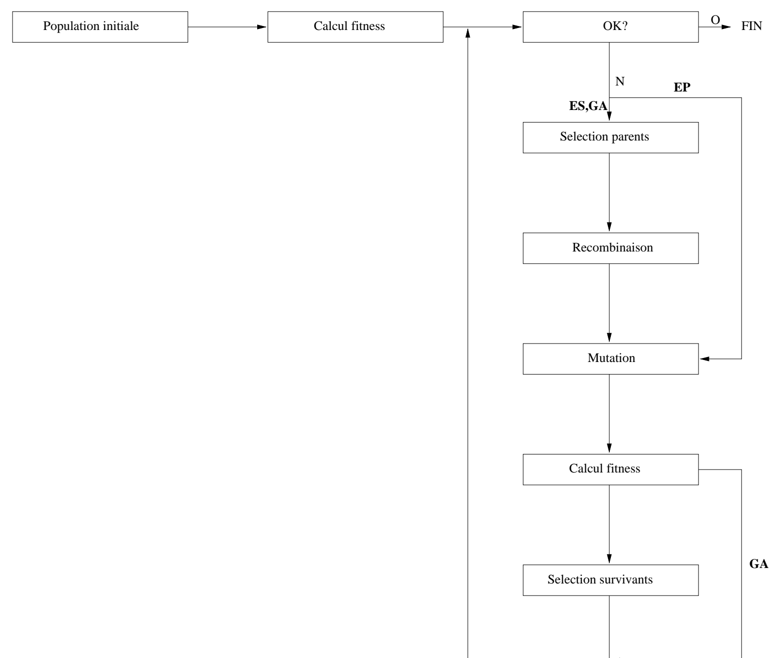

FIG. 1.13 – *Phases des différentes méthodologies [1].*

1.3.1 Principes de base

Encodage des données

Initialement, la théorie des algorithmes évolutionnaires a été établie pour des génotypes sous forme binaire. Comme nous le verrons par la suite, ce type de données permet de s'assurer d'une certaine convergence. Cependant, au fil du temps, les opérations des algorithmes évolutionnaires ont été étendues aux variables décimales, réelles ou quelconques. Elles permettent ainsi d'obtenir les avantages de ces algorithmes sans devoir effectuer en encodage sous forme binaire. Nous revenons sur ces extensions par la suite.

Evaluation

Lors de l'évaluation des différents candidats, il est nécessaire de disposer d'une bonne fonction de performance, aussi appelée *fitness function*, afin de pouvoir ordonner ceux-ci de manière optimale. Par exemple, pour un problème de maximisation, nous classerons les candidats en fonction de la valeur de la fonction objectif, qui est la mesure la plus simple de la qualité d'un candidat.

Sélection des parents

Un algorithme évolutionnaire doit faire un bon compromis entre l'aspect **exploitation** et **exploration**. Cette phase de l'algorithme favorise l'aspect *exploitation* de l'espace de recherche. C'est pourquoi certaines méthodologies négligent celle-ci (EP).

Cette opération de sélection peut être effectuée dans la population des candidats. Une telle sélection permet d'obtenir les futurs parents de la prochaine population. Il existe de nombreuses méthodes de sélection parentale. Deux grandes classes existent :

La sélection déterministe : Les parents sélectionnés sont les candidats les mieux adaptés. Les individus choisis sont donc les meilleurs candidats au sens de la *fitness function*.

La sélection stochastique : Les parents sont sélectionnés de manière stochastique. Ainsi, de plus *faibles* individus peuvent être choisis au détriment des plus *forts*.

Lorsque les parents sont sélectionnés, d'autres opérations doivent encore être effectuées afin de disposer de la population suivante.

Recombinaison

La recombinaison (ou *cross-over*) est l'opération qui consiste à obtenir un (ou plusieurs) nouveau(x) candidat(s) à l'aide de parent(s) sélectionné(s) dans la population (éventuellement réduite suite à la sélection des parents).

A l'aide de deux suites de bits, nous créons deux nouveaux individus :
- L'un composé de la première partie de la première suite et de la seconde partie de la deuxième suite;
- L'autre composé de la seconde partie de la première suite et de la première partie de la deuxième suite.

En fonction des besoins, cette recombinaison est effectuée avec une plus ou moins forte probabilité. Ce choix peut être effectué en fonction de la nature des individus à croiser (faible ou fort au sens de la *fitness function*).

Mutation

La mutation est l'opération qui consiste à modifier un ou plusieurs bits d'un individu de façon aléatoire. Cette opération est effectuée avec une faible probabilité. Elle a pour but de créer ou recréer une valeur absente d'un certain gène.

Sélection des survivants

Afin d'obtenir un algorithme évolutionnaire, il nous faut procéder à une dernière étape avant la prochaine itération de l'algorithme. Pour obtenir une population de taille équivalente à la précédente, il faut sélectionner les individus qui en feront partie. Cette opération peut se faire de plusieurs manières différentes :

Pure reinsertion : Tous les enfants remplacent leurs parents. Ainsi, chaque individu ne *vit* que pendant une itération de l'algorithme. Il faut veiller à produire un nombre suffisant d'enfants pour remplacer les candidats de la population précédente.

Uniform reinsertion : Les enfants produits remplacent les individus de la population précédente de manière aléatoire.

Elitist reinsertion : Les enfants produits remplacent les moins bons individus de la population précédente.

Fitness-based reinsertion : Plus d'enfants que nécessaire sont produits lors de la phase de recombinaison. Ensuite, seuls les enfants les plus performants sont retenus pour faire partie de la population suivante.

Une bonne combinaison de ces différentes techniques permet de produire une population d'individus de plus en plus performants afin d'obtenir une bonne convergence de l'algorithme évolutionnaire. Ainsi, si les meilleurs enfants remplacent les moins bons parents, la population ne cesse de s'améliorer jusqu'à l'obtention d'une solution souhaitée au problème posé.

Arrêt de l'algorithme

Comme dans tout algorithme, il est important de choisir un bon critère d'arrêt. Dans le cas des algorithmes évolutionnaires, le choix le plus simple est le nombre d'itérations. Cependant, il est également possible d'analyser les populations générées afin de stopper l'algorithme lorsque les individus cessent d'évoluer. Dans la section suivante, nous présentons quelques éléments théoriques permettant d'analyser ces populations et d'en déduire une certaine convergence.

1.3.2 Eléments théoriques

Les algorithmes évolutionnaires sont des méthodes heuristiques ayant fait leurs preuves mais dont aucun théorème quant à une éventuelle convergence n'a pu être démontré. Cependant, il existe certains résultats permettant d'analyser l'évolution d'un type d'individus dans une population. Les résultats présentés dans cette section sont inspirés de [22] et [3].

Définitions

Afin de présenter les différents résultats concernant les opérations des algorithmes évolutionnaires, nous allons définir quelques notions.

Définition 1.2 (Séquence) *Une séquence S de longueur n est une suite*

$$A = a_1 a_2 \ldots a_n \quad \textit{où} \quad a_i \in V = \{0{,}1\} \quad \forall\, i = 1..n$$

Définition 1.3 (Schéma) *Un schéma H de longueur n est une suite*

$$H = a_1 a_2 \ldots a_n \quad \textit{où} \quad a_i \in V^+ = \{0{,}1{,}*\} \quad \forall\, i = 1..n$$

Le terme $*$ de l'ensemble V^+ signifie que l'élément s_i peut prendre indifféremment la valeur 0 ou la valeur 1.

Définition 1.4 (Instance) *Une séquence $A = a_1 \ldots a_n$ est une instance du schéma $H = b_1 \ldots b_n$ si pour tout i tel que $b_i \neq *$, on a $a_i = b_i$.*

Illustrons ces définitions par un petit exemple. Si nous considérons le schéma $0011*101$, les seules instances possibles de celui-ci sont les séquences 00110101 et 00111101.

Définition 1.5 (Position fixe, position libre) *La position i est dite fixe dans un schéma H si $a_i = 0$ ou $a_i = 1$. La position i est dite libre dans un schéma H si $a_i = *$.*

Définition 1.6 (Ordre d'un schéma) *L'ordre d'un schéma H, noté $o(H)$, est le nombre de positions fixes de H.*

Définition 1.7 (Longueur fondamentale) *La longueur fondamentale d'un schéma H, notée $\delta(H)$, est la distance séparant la première position fixe de H de la dernière position fixe de H.*

Définition 1.8 (Adaptation (*fitness*) d'une séquence) *L'adaptation d'une séquence A est une valeur positive notée $f(A)$.*

Dans les algorithmes évolutionnaires, l'adaptation d'une séquence est équivalente à la *fitness* d'un élément de la population.

Définition 1.9 (Adaptation (*fitness*) d'un schéma) *Un schéma H possède une adaptation valuée à*

$$f(H) = \frac{\sum_{i=1}^{2^{n-o(H)}} f(A_i)}{2^{n-o(H)}}$$

où les A_i représentent l'ensemble des instances de H.

A l'aide de ces quelques définitions, nous allons analyser les effets des différentes opérations effectuées sur la population dans une itération d'une algorithme évolutionnaire.

Sélection des parents

Considérons un ensemble constitué de n séquences de bits constituant la population à un moment donné

$$S = \{A_1, \ldots, A_i, \ldots, A_n\}.$$

La probabilité de survie d'un élément de cet ensemble dépend de la *fitness* (ou adaptation) de celui-ci :

$$p_i = \frac{f(A_i)}{\sum_{i=1}^{n} f(A_i)}.$$

Nous allons introduire une nouvelle valeur, à savoir le nombre d'instances d'un schéma H dans une population à un moment donné que nous allons noter $m(H,t)$. L'analyse de cette valeur nous permettra d'obtenir un résultat intéressant quant à la convergence des algorithmes évolutionnaires.

Le nombre d'instances d'un schéma H à l'instant $t+1$ évolue en fonction de la *fitness* (ou adaptation) du schéma H dans la population à l'instant t. Ainsi, nous obtenons le résultat suivant :

$$m(H,t+1) = m(H,t).n.\frac{f(H)}{\sum_{i=1}^{n} f(A_i)}. \tag{1.1}$$

Afin de pouvoir interpréter de manière plus claire le résultat, nous allons simplifier sa notation en définissant de nouvelles valeurs en fonction des autres.

Posons

$$\bar{f}_t = \frac{\sum_{i=1}^{n} f(A_i)}{n}$$

Ce nombre représente la moyenne de l'adaptation des différents éléments de la population à l'instant t. La formule 1.1 se réécrit sous la forme suivante :

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}_t}. \tag{1.2}$$

Posons ensuite $c_t(H) = \frac{f(H)}{\bar{f}_t} - 1$. Le résultat 1.2 peut encore se réécrire en fonction de cette nouvelle valeur comme

$$m(H,t+1) = (1 + c_t(H))m(H,t) \tag{1.3}$$

Nous pouvons maintenant interpréter le résultat de façon plus claire. En effet, nous constatons que lorsqu'un schéma est mieux adapté que l'ensemble des éléments de la population à l'instant t (au sens de la moyenne), son nombre de représentants (instances) dans la population à l'instant $t+1$ augmente.

De plus, si nous faisons l'approximation que $c_t(H)$ est constant au fil du temps, cette augmentation du nombre d'instances d'un schéma dans la population suit une loi géométrique, i.e.

$$m(H,t) = (1 + c(H))^t.m(H,0).$$

Ainsi, nous pouvons donc conclure que plus un schéma est fort (au sens de la fonction *fitness* ou d'adaptation), plus le nombre d'instances de celui-ci augmentera au fil des itérations de l'algorithme.

Recombinaison

Lors d'une opération de recombinaison, un schéma H peut voir son nombre de représentants diminuer. C'est pourquoi nous devons analyser la probabilité pour que le schéma H survive suite à cette recombinaison. Cette probabilité, notée $p_s(H)$, ne peut être qu'approximée car elle est fonction de la probabilité de croisement (p_c) du schéma H avec un autre schéma. De plus, elle dépend également du nombre de positions libres du schéma H. Afin d'obtenir une borne inférieure sur cette probabilité de survie de façon plus concise, nous allons utiliser la longueur fondamentale de H, au lieu de

spécifier toutes ses positions libres.

Ainsi, nous avons une borne inférieure sur la probabilité de détruire un schéma H, à savoir $\delta(H)/(l-1)$. En introduisant la probabilité de recombinaison, nous obtenons le résultat suivant sur la probabilité de survie du schéma H :

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1}$$

En combinant cette borne inférieure au résultat 1.3, nous obtenons une évolution du nombre d'instances du schéma H de la forme :

$$m(H,t+1) \geq m(H,t)(1+c_t(H)) \left(1 - p_c \frac{\delta(H)}{l-1}\right) \tag{1.4}$$

Mutation

Lors d'une mutation sur un schéma H, seules les positions fixes peuvent être détruites. Ainsi, la probabilité de survie du schéma H dans la population sera liée à l'ordre de H, $o(H)$. Si la probalité de mutation est notée p_m et que celle-ci est très faible (de l'ordre de 0.001), nous pouvons obtenir la probabilité de survie de H et développer celle-ci au premier ordre, i.e.

$$p_s = (1-p_m)^{o(H)} \cong 1 - o(H)p_m$$

Le résultat 1.4 s'énonce donc de la manière suivante suite à l'introduction de cette nouvelle donnée :

$$m(H,t+1) \geq m(H,t)(1+c_t(H)) \left(1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m\right)$$

Ce dernier résultat constitue le *Théroème des schémas* tel que démontré par Goldberg dans [22].

Discussion

En analysant le résultat obtenu, nous pouvons faire deux constatations :

- Les schémas plus forts (au sens de la fonction d'adaptation) sont favorisés pour la génération de la nouvelle population.

- Les schémas ayant une longueur fondamentale plus petite que les autres sont favorisés pour la génération de la nouvelle population;

- Les schémas ayant un ordre plus petit que les autres sont favorisés pour la génération de la nouvelle population.

Il est important de tenir compte de ces remarques lors de la modélisation d'un problème. En effet, l'encodage des données sous forme de séquences de bits doit se faire en respectant certaines règles :

- Les données censées être plus proche de la solution doivent être représentées par des instances d'un schéma dont la longueur fondamentale et l'ordre sont petits.

- Les données censées être *sans intérêt* doivent être représentées par des instances d'un schéma dont la longueur fondamentale et l'ordre sont élevés.

Un tel encodage n'est pas toujours facile à mettre en place car il nécessite une bonne connaissance du problème à résoudre.

1.3.3 Extensions

Les résultats que nous venons de citer n'ont pu, jusqu'à présent, être énoncés que pour des données encodées sous forme de séquences de bits. Cependant, il n'est pas toujours possible d'avoir un tel encodage.

Considérons le problème de programmation mathématique suivant :

$$\begin{array}{lll} \min & f(x) & \\ \text{s.c.} & g_i(x) \leq 0 & i = 1, \ldots, m \\ & h_j(x) = 0 & j = 1, \ldots, p \end{array}$$

où $x \in \mathbb{R}^n$.

Il existe de nombreux algorithmes permettant d'obtenir un minimum local de ce problème. Ces techniques sont principalement basées sur les dérivées successives de la fonction objectif. La solution trouvée peut être locale, c'est-à-dire qu'elle est minimale seulement dans un intervalle autour d'elle. L'application des algorithmes évolutionnaires à ce type de problèmes permet de se baser uniquement sur la valeur de la fonction objectif et d'effectuer plusieurs recherches dans l'espace des contraintes, comme nous le montre la figure 1.14. Les points noirs représentent les individus de la population et la courbe, la fonction à minimiser.

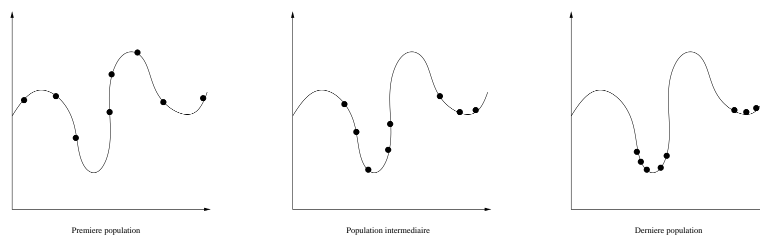

FIG. 1.14 – *Exemple d'utilisation d'algorithme évolutionnaire sur un problème mathématique.*

Le principal intérêt de ces algorithmes appliqués aux problèmes mathématiques réside dans le fait qu'ils permettent de trouver une **solution globale** là où d'autres algorithmes ne proposeraient qu'une solution locale.

Cependant, pour de tels types de problèmes, il est impossible d'encoder les individus de la population sous forme binaire. En effet, il n'existe au-

cun isomorphisme entre $\mathbb{R}^n$ et $\{0,1\}^n$. L'algorithme évolutionnaire est donc utilisé sans encoder les données en séquences de bits. Les opérateurs de recombinaison et de mutation doivent être exprimés différemment.

Recombinaison

En ce qui concerne la recombinaison, le principe est de croiser deux individus afin d'en générer d'autres ayant certaines caractéristiques des deux parents.

Recombinaison discrète :
La valeur d'une variable de l'enfant est obtenue en sélectionnant de manière aléatoire la valeur de cette variable chez un parent. Par exemple, si l'on considère les 2 parents représentés par les vecteurs (31 6 75 15) et (54 9 34 5), un des enfants pourrait être (54 6 34 15).

Recombinaison intermédiaire :
Dans le cas de données réelles, l'enfant peut être une combinaison linéaire des deux parents, i.e.

$$enfant = parent1 + \alpha(parent2 - parent1)$$

où α est un vecteur choisi de manière aléatoire dans l'intervalle $[-d,1+d]^n$. Si la valeur de d est nulle, cette recombinaison est dite intermédiaire, sinon elle est caractérisée de recombinaison intermédiaire *étendue*.

Mutation

En ce qui concerne la mutation, il s'agit de perturber une ou plusieurs variables du vecteur original. Cette opération est effectuée par l'ajout d'un nombre aléatoire très petit.

Discussion

Tout comme pour l'algorithme original, ces opérations sont effectuées avec une faible probabilité. Cependant, il n'existe aucune théorie permettant de s'assurer d'une certaine convergence d'un algorithme évolutionnaire appliqué à un problème à variables réelles.

L'expérience a permis de constater que cette extension de l'algorithme évolutionnaire original obtenait généralement de meilleurs résultats et ce, plus rapidement.

Chapitre 2

Apprentissage automatique

Dans le chapitre précédent, nous avons présenté diverses méthodes afin de permettre à un ordinateur de prendre une décision. Cependant, autant pour les réseaux de neurones que pour les algorithmes évolutionnaires, il est préférable d'avoir une bonne connaissance du problème à résoudre afin d'effectuer un meilleur choix (poids, paramètres de recombinaison ou de mutation). Cette connaissance peut être obtenue suite à un apprentissage par l'ordinateur. Cette phase de calibration est effectuée grâce à une série d'exemples permettant d'avoir une vision plus ou moins globale de la classe de problèmes à résoudre.

Dans ce chapitre, nous commençons par une introduction basée sur des raisonnements logiques pouvant être effectués par des ordinateurs. Ensuite, nous présentons la manière dont nous pouvons entrainer des réseaux de neurones afin de les rendre capable de résoudre une certaine classe de problèmes.

2.1 Induction - Déduction

Afin de raisonner de façon cohérente, un ordinateur peut faire appel à différentes méthodes que sont l'induction ou la déduction. Les définitions suivantes sont tirées de [41].

Définition 2.1 (Induction) *Généralisation d'une observation ou d'un raisonnement établis à partir de cas singuliers.*

Définition 2.2 (Déduction) *Conséquence tirée d'un raisonnement.*

Un exemple d'induction peut être un raisonnement de ce type :

Tous les chats que j'ai rencontrés étaient gris.
Donc tous les chats sont gris.

Evidemment, cette affirmation ne peut pas être prouvée mais elle constitue une règle suffisamment raisonnable en absence de contre-exemples.

Quant à la déduction, elle constitue un raisonnement logique correct. Par exemple,

Tous les hommes sont mortels.
Socrate est un homme.
Donc Socrate est mortel.

La combinaison de ces deux raisonnements peut permettre à un ordinateur d'obtenir des résultats cohérents face à des problèmes particuliers. Il procède premièrement à un apprentissage à l'aide d'exemples (induction) pour ensuite raisonner par déduction afin de fournir une solution censée être correcte.

Afin d'illustrer ces principes, nous présentons un exemple basé sur les perceptions météorologiques et la pratique d'un sport. Cet exemple est tiré de [2].

Pour pratiquer son sport, une personne analyse le temps qu'il fait. Son choix dépend de plusieurs paramètres tels que l'humidité, le vent, la luminosité ou la visibilité. Le tableau 2.1 présente une liste reprenant les choix effectués par cette personne en fonction de ces différents paramètres. Nous utilisons ces exemples pour calibrer l'algorithme.

Sky	Temp	Humid	Wind	Water	ForeCast	EnjoySport
Sunny	Warm	Normal	Strong	Warm	Same	Yes
Sunny	Warm	High	Weak	Warm	Same	Yes
Rainy	Cold	High	Strong	Warm	Change	No
Sunny	Warm	High	Strong	Cool	Change	Yes

TAB. 2.1 – *Ensemble d'exemples pour le calibrage d'un algorithme de déduction.*

Afin de pouvoir définir un algorithme sur base de ces exemples, il est nécessaire de les encoder sous forme de *n-uplet* reprenant les différentes caractéristiques météorologiques :

$$\begin{array}{cccccccc} & & \text{Sky} & \text{Temp} & \text{Humid} & \text{Wind} & \text{Water} & \text{ForeCast} & \\ x = & < & Sunny & Warm & Normal & Strong & Warm & Same & > \end{array}$$

Une fonction de sortie est associée à chacun de ces exemples afin de savoir si le sport sera pratiqué :

$$c : X \rightarrow \{0,1\} : x \leadsto c(x)$$

où $c(x)$ vaut 0 si le sport n'est pas pratiqué, 1 si le sport est pratiqué
et X est l'ensemble des possibilités météorologiques.

Afin de pouvoir effectuer un raisonnement correct, l'algorithme utilise une hypothèse de déduction, c'est-à-dire un ensemble de contraintes que doit satisfaire plusieurs caractéristiques météorologiques afin que la personne pratique son sport. Nous utilisons une notation sous la forme d'un *n-uplet* pour simplifier l'utilisation des contraintes d'une telle hypothèse. Celles-ci peuvent prendre des formes différentes selon la restriction à établir :

- Valeur spécifique : $Sky = Sunny$
- Valeur sans importance : $Sky = ?$
- Valeur interdite : $Sky = \otimes$

Si une contrainte de type *"Valeur Interdite"* apparaît dans une hypothèse, celle-ci sera insatisfaisable quelle que soit la valeur de la caractéristique météorologique correspondante.

Les contraintes $(Sky = Sunny)$ et $(Sky = Rainy)$ ne peuvent être comparées. Par contre, un ordre peut être établi entre certaines restrictions, e.g.

$$(Sky = \otimes) < (Sky = Sunny) < (Sky = ?)$$

Ces inégalités définissent une relation d'ordre partiel permettant de classer les contraintes. Ainsi, nous dirons qu'une contrainte x est plus particulière qu'une autre y si elle respecte la relation $x < y$ telle que nous l'avons définie ci-dessus. En généralisant cet ordre, nous pouvons obtenir un ordre partiel sur l'espace des hypothèses.

Si nous considérons

$$\begin{array}{ccccccccc} & & \text{Sky} & \text{Temp} & \text{Humid} & \text{Wind} & \text{Water} & \text{ForeCast} & \\ h = & < & Sunny & Warm & ? & ? & Warm & Same & > \end{array}$$

nous pouvons constater que les lignes 1 et 2 du tableau 2.1 satisfont les contraintes de cette hypothèse, tandis que la ligne 3 et 4 ne les satisfont pas.

Dans un ensemble d'hypothèses H, nous pouvons retrouver deux hypothèses particulières :

$$h_G = (?,?,?,?,?,?) \tag{2.1}$$

$$h_S = (\otimes, \otimes, \otimes, \otimes, \otimes, \otimes) \tag{2.2}$$

Les contraintes de l'hypothèse h_G sont toujours satisfaites, tandis que les contraintes de l'hypothèse h_S ne le sont jamais. De plus, nous avons

$$\forall\, h \in H,\; h_S \leq h \leq h_G$$

Dans le cas où l'hypothèse h_S serait retenue pour un algorithme de déduction, cela signifierait que le sport ne sera pratiqué en aucun cas. A l'opposé, si l'hypothèse h_G est retenue, le sport sera pratiqué quelles que soient les conditions météorologiques.

Nous pouvons classer les éléments d'un ensemble d'exemples D (e.g. Tableau 2.1) dans deux ensembles par rapport à une hypothèse h :

1. D'un côté, les éléments dont les caractéristiques satisfont les contraintes de h.
2. De l'autre côté, les éléments dont les caractéristiques ne respectent pas les contraintes de h.

A l'aide de ces deux ensembles, nous avons la possibilité d'associer une fonction à l'hypothèse h

$$h : X \rightarrow \{0,1\} : x \rightsquigarrow h(x)$$

où $h(x)$ vaut 1 si x est compatible avec h, 0 sinon.

En utilisant les notions présentées ci-dessus, nous pouvons exprimer la phase d'apprentissage de la façon suivante :

Phase d'apprentissage :

Etant donné

- L'ensemble des possibilités X caractérisées par la valeur des attributs Sky,$Temp$,$Humid$,$Wind$,$Water$,$ForeCast$
- Une fonction résultat $c : X \rightarrow \{0,1\}$
- Un ensemble d'hypothèses H
- Un ensemble D d'exemples d'entrainement positifs et négatifs de la fonction résultat
  $< x_1,c(x_1) > , \ldots , < x_m,c(x_m) >$

On désire obtenir une hypothèse $h \in H$ telle que
$h(x) = c(x) \ \forall \ x \in D$

A présent, nous devons trouver une méthode afin d'obtenir le résultat que nous venons de présenter. Une première solution consiste à obtenir une hypothèse d'acceptation en fonction des exemples positifs. De façon formelle, nous obtenons l'algorithme suivant :

Algorithme d'apprentissage :

1. Initialiser h à l'hypothèse la plus particulière de H (cfr équation 2.2)

2. **Pour** tous les exemples positifs $d \in D$,
   Pour chaque contrainte a_i dans h,
   Si la contrainte a_i dans h est satisfaite par d
   Alors ne rien faire
   Sinon remplacer a_i dans h par la contrainte
   la plus particulière que satisfait d

3. L'hypothèse h obtenue est l'hypothèse recherchée.

Cet algorithme possède un inconvénient car il ne permet de disposer que d'une seule hypothèse de déduction. Afin de pouvoir écrire un algorithme plus performant, nous allons définir de nouveaux concepts.

Une hypothèse h est dite **consistante** avec un ensemble d'exemples D, auquel est associé une fonction de sortie c, si et seulement si $h(x) = c(x)$ pour tous les exemples $< x,c(x) >$ dans D, i.e.

$$Consistent(h,D) \Leftrightarrow \forall \ < x,c(x) > \in D : h(x) = c(x)$$

Un **espace de version** par rapport à l'espace d'hypothèses H et l'ensemble d'exemples D, noté $VS_{H,D}$ est le sous-ensemble d'hypothèses de H consistantes avec D, i.e.

$$VS_{H,D} \equiv \{h \in H | Consitent(h,D)\}$$

Sous sa forme simplifiée, le nouvel algorithme proposé s'exprime de la façon suivante :

Algorithme d'apprentissage (seconde version) :

1. $VS \leftarrow$ une liste de toutes les hypothèses de H

2. **Pour** tous les exemples $< x,c(x) > \in D$,
   Retirer de VS toutes les hypothèses h telles que $h(x) \neq c(x)$

3. L'algorithme nous fournit la liste des hypothèses consistantes avec D.

Le principal inconvénient de cet algorithme s'avère être la taille de l'ensemble H. En effet, pour chaque exemple, nous devons parcourir l'espace tout entier, ce qui peut être pénalisant. Afin d'éviter ce problème, nous allons nous servir de deux ensembles d'hypothèses nous permettant au final de générer l'ensemble des hypothèses consistantes avec D.

Soient G, l'ensemble des hypothèses les plus générales de $VS_{H,D}$, et S, l'ensemble des hypothèses les plus particulières (ou spécifiques) de $VS_{H,D}$, tout l'espace de version $VS_{H,D}$ est contenu entre ces deux bornes, i.e.

$$VS_{H,D} = \{h \in H \mid \exists\, s \in S, \exists\, g \in G : s \leq h \leq g\}$$

où $x \leq y$ signifie que x est une hypothèse plus particulière que (ou équivalente à) y.

A l'aide de l'algorithme suivant, nous allons spécifier les deux ensembles d'hypothèses nous permettant de générer uniquement les hypothèses consistantes avec l'ensemble d'exemples D.

Algorithme d'apprentissage (troisième version) :

$G \leftarrow$ hypothèse la plus générale de H
$S \leftarrow$ hypothèse la plus particulière de H
Pour chaque exemple $d \in D$
    Si d est un exemple positif,
        - Retirer de G les hypothèses incompatible avec d
        - **Pour** chaque hypothèse $s \in S$ incompatible avec d,
            • Retirer s de S
            • Ajouter à S les généralisations minimales h de s tel que
                * h est compatible avec d
                * une hypothèse de G est plus générale que h
            • Retirer de S toutes les hypothèses plus générales que toute autre hypothèse de S
    Si d est un exemple négatif,
        - Retirer de S les hypothèses incompatibles avec d
        - **Pour** chaque hypothèse $g \in G$ incompatible avec d,
            • Retirer g de G
            • Ajouter à G les particularisations minimales h de g tel que
                * h est compatible avec d
                * une hypothèse de S est plus spécifique que h
            • Retirer de G toutes les hypothèses plus spécifiques que toute autre hypothèse de G

Illustrons cet algorithme à l'aide du tableau d'apprentissage 2.1 présenté au début de cette section.

Au départ de l'algorithme, les ensembles G et S ne sont composés que d'une seule hypothèse, i.e.

$$G = \{h_G\}$$

$$S = \{h_S\}$$

où h_G et h_S sont les hypothèses définies par les équations 2.1 et 2.2.

A l'aide du premier exemple, les deux ensembles sont mis à jour afin d'obtenir les résultats suivants

$$G = \{h_G\}$$
$$S = \{(Sunny,Warm,Normal,Strong,Warm,Same)\}$$

Ensuite, le deuxième exemple permet d'affiner l'ensemble S, i.e.

$$G = \{h_G\}$$
$$S = \{(Sunny,Warm,?,?,Warm,Same)\}$$

Le troisième exemple (négatif) est intéressant car il entraîne des modifications dans l'ensemble G alors que, jusqu'à présent, les transformations ont été appliquées à l'ensemble S:

$$G = \{(Sunny,?,?,?,?,?),(?,Warm,?,?,?,?),(?,?,?,?,?,Same)\}$$
$$S = \{(Sunny,Warm,?,?,Warm,Same)\}$$

Finalement, le dernier exemple nous permet d'obtenir les ensembles suivants

$$G = \{(Sunny,?,?,?,?,?),(?,Warm,?,?,?,?)\}$$
$$S = \{(Sunny,Warm,?,?,?,?)\}$$

En analysant l'espace que nous pouvons engendrer avec ces deux ensembles, nous constatons que seuls leurs éléments peuvent être générées. Ainsi, l'hypothèse de déduction de l'algorithme devra être choisie parmi ceux-ci pour obtenir une hypothèse consistante avec l'ensemble d'exemples présenté.

Cette forme d'apprentissage paraît simpliste mais elle peut être suffisante pour une certaine classe de problèmes. Quand la situation ne le permet pas, l'hypothèse de déduction peut être remplacée par un réseau de neurones capable de raisonner de la même façon suite à un apprentissage. Cette phase de calibration fait l'objet de la section suivante.

2.2 Réseaux de neurones

Dans le cadre des réseaux de neurones, deux formes d'apprentissage sont utilisées. Nous présentons premièrement une technique supervisée basée sur une série d'exemples. Ensuite, nous montrons qu'il est possible d'utiliser une technique afin que le réseau apprenne totalement de lui-même.

2.2.1 Apprentissage supervisé

L'apprentissage supervisé se base sur des exemples pour lesquels les résultats souhaités sont connus. Cela permet de comparer la sortie effective générée par un réseau à la sortie espérée et de corriger le réseau pour qu'il se rapproche du comportement souhaité. Les méthodes d'apprentissage supervisé nécessitent de choisir un ensemble d'exemples suffisamment variés. Si ce n'est pas le cas, le réseau peut être *sur-adapté* à un sous-ensemble de situations et se comporter de manière insatisfaisante dans les autres cas.

La règle de Hebb

La règle de Hebb est basée sur une analogie avec les réseaux de neurones biologiques : Dans un système nerveux, les connexions se renforcent dans les zones où l'activité est importante.

Pour appliquer cette règle sur un réseau de neurones artificiels, une entrée est soumise au réseau, ce qui a pour effet d'activer certains neurones. Comme la sortie désirée est connue, les neurones de sortie supposés actifs sont également connus. La règle de Hebb consiste à augmenter le poids des arcs reliant les neurones actifs, c'est-à-dire augmenter $p_{ij} \in [0,1]$ si les neurone i et j sont actifs.

Cette méthode telle que proposée par Hebb possède un inconvénient. En effet, si au fil des exemples, les poids des arcs sont continuellement augmentés, le réseau devient totalement saturé, ce qui n'est pas souhaitable.

Pour éliminer cet inconvénient, Wulfram Gerstner et Werner M. Kistler proposent dans [43] de mettre en place un système qui aténue progressivement tous les poids à chacun des apprentissages. Cette version améliorée de la règle de Hebb peut être formulée comme suit :

$$\Delta p_{ij} = \lambda_1(1 - p_{ij})a_i a_j - \lambda_2 p_{ij}$$

où a_i vaut 1 si le neurone i est actif, zéro sinon. Les valeurs λ_1 et λ_2 permettent de paramétrer respectivement le taux de croissance et de décroissance des poids.

La valeur de Δp_{ij} permet de mettre à jour le poids de chacun des arcs du réseau. Tous les poids sont décrémentés proportionnellement à leurs valeurs et à λ_2. Les arcs reliant deux neurones a_i et a_j actifs sont incrémentés proportionnellement à $1 - p_{ij}$. Cette formulation crée un mécanisme de convergence asymptotique vers les bornes : un poids élevé reliant des neurones non-actifs lors d'un apprentissage sera fortement décrémenté alors qu'un poids faible n'est décrémenté que légèrement. De même, un poids élevé reliant des neurones actifs sera peu incrémenté et un poids faible sera fortement incrémenté.

Règle du delta rétro-propagée

Cette méthode d'apprentissage introduite indépendamment par plusieurs chercheurs est actuellement la plus répandue. L'idée de base est de comparer la sortie générée par le réseau sur un exemple d'apprentissage à la sortie souhaitée et de modifier les poids du réseau pour minimiser l'erreur commise par le réseau.

Dans le cas du modèle à simple couche, la règle du delta met à jour les poids des arcs en utilisant le gradient de l'erreur commise par le réseau. Nous présentons ci-dessous ce critère sous sa forme mathématique.

Soient x_j la valeur en sortie du neurone j, x_j^* la sortie désirée de ce même neurone et s_j le résultat de la fonction d'entrée, c'est-à-dire $\sum_{i \in I} p_{ij} x_i$ où $I = \{i \text{ tq } x_i \text{ est un neurone d'entrée}\}$. Si $g(\cdot)$ est la fonction d'activation du neurone, alors $x_j = g(s_j)$.

Supposons que l'erreur totale, désignée par E, est la somme des carrés des erreurs commises sur chaque valeur de sortie :

$$E = \sum_{j \in J} (x_j^* - x_j)^2, \tag{2.3}$$

où $J = \{j \text{ tq } x_j \text{ est un neurone de sortie}\}$.

Nous désirons exprimer le gradient de l'erreur. En dérivant l'erreur par rapport au poids p_{ij}, nous obtenons

$$\frac{\partial E}{\partial p_{ij}} = -2(x_j^* - x_j)\frac{\partial x_j}{\partial p_{ij}}$$

Or, sachant que l'identité suivante est vraie

$$\frac{\partial x_j}{\partial p_{ij}} = \frac{\partial x_j}{\partial s_j}\frac{\partial s_j}{\partial p_{ij}}$$

où

$$\frac{\partial x_j}{\partial s_j} = \frac{\partial g(s_j)}{\partial s_j} = g'(s_j)$$

et

$$\frac{\partial s_j}{\partial p_{ij}} = \frac{\partial\left(\sum_j p_{ij}x_j\right)}{\partial p_{ij}} = x_i\,,$$

nous obtenons l'expression finale suivante pour (2.3)

$$\frac{\partial E}{\partial p_{ij}} = -2(x_j^* - x_j)g'(s_j)x_i$$

qui est la composante suivant p_{ij} du gradient de l'erreur (exprimée dans l'espace des poids p_{ij}) commise par le réseau de neurones. Un pas dans la direction de la plus forte pente permet de minimiser cette erreur. L'expression de la plus forte pente est

$$\Delta p_{ij} = \lambda(x_j^* - x_j)g'(s_j)x_i$$

où λ est une constante qui permet de jouer sur la convergence de la méthode.

Dans le cas particulier d'ADALINE (cfr §1.2.2), cette dernière formule s'écrit sous la forme

$$\Delta p_{ij} = \lambda(x_j^* - x_j)x_i$$

Si le réseau est un réseau multi-couches, les poids des arcs connectés aux couches internes doivent être également mis à jour. Pour cela, il faut estimer l'erreur commise à la sortie des neurones internes. Cette estimation se fait en *rétro-propageant* l'erreur commise par les neurones de sortie vers la couche

intérieure. Soit la couche k pour laquelle l'erreur en sortie est connue, alors l'erreur en sortie des neurones de la couche $k-1$ se calcule comme suit :

$$E_i = \sum_{j \in J} p_{ij} E_j g'(s_i) \quad \text{où } i \in I \tag{2.4}$$

$$I = \{i \text{ tq } x_i \text{ appartient à la couche } k-1\}$$
$$J = \{j \text{ tq } x_j \text{ appartient à la couche } k\}$$

L'équation (2.4) devient alors, si la couche k est la couche de sortie :

$$E_i = \sum_{j \in J} p_{ij}(x_j^* - x_j) g'(s_i)$$

Cette *rétro-propagation* de l'erreur permet d'appliquer la règle du delta sur les poids internes au réseau.

Convergence de la méthode

Pour converger correctement, la méthode nécessite de choisir judicieusement la valeur du λ de la règle du delta. Si celle-ci est trop petite, la méthode va converger très lentement. Si elle est trop grande les déplacements dans l'espace des poids risquent d'être trop ératiques et d'empêcher toute convergence.

Pour éviter cette instabilité, une inertie est introduite dans les déplacements effectués. Le calcul du pas dans l'espace des poids est influencé par le pas précédent:

$$\Delta p_{ij}^{t_1} = \lambda(x_j^* - x_j)x_i + \varepsilon \Delta p_{ij}^{t_0}$$

où $\Delta p_{ij}^{t_1}$ est le pas à calculer, $\Delta p_{ij}^{t_0}$ est la valeur du pas prédédent et ε une petite constante.

Une autre manière d'améliorer la convergence consiste à accumuler les erreurs en soumettant une série de couples *entrées-sorties* espérées. Pour chacun des noeuds de sorties, les erreurs commises par le réseau sont accumulées avant d'appliquer l'algorithme de rétro-propagation.

Au delà du choix du paramètre λ lors du calcul du Δp, le passage d'un réseau simple-couche à un réseau multi-couches apporte d'autres difficultés. En effet pour avoir un sens, un tel réseau doit utiliser des fonctions d'activation non-linéaires. Dès lors, la fonction d'erreur E est également non-linéaire et peut donc présenter des minima locaux qui mettent en péril la méthode du gradient de descente.

Un autre inconvénient dont souffre la méthode du gradient de descente apparaît quand la surface définie par la fonction d'erreur présente un plateau. Dans ce cas le gradient est quasi nul, la mise à jour des poids piétine et provoque une situation de blocage. Par exemple, des neurones utilisant des fonctions sigmoïdes comme fonctions d'activation vont générer un gradient proche de zéro si les poids p_{ij} croissent vers des valeurs trop élevées.

Au delà de ces critiques techniques, il ne faut pas perdre de vue que si les réseaux de neurones rétro-propagés s'appuient sur des concepts biologiques, ils ne sont finalement que des supports permettant de modéliser des formules mathématiques. De plus, la méthode du delta n'est rien d'autre que l'application d'un gradient de descente qui est une des méthode les plus triviales en optimisation numérique.

2.2.2 Apprentissage non supervisé

L'apprentissage non supervisé est utilisé quand aucun exemple d'entrainement n'est disponible. C'est le cas notamment de réseaux de neurones dont les résultats sont comparables de manière relative mais pour lesquels aucune référence absolue n'est disponible.

Dans ce type de situations, la règle du delta rétro-propagé est inapplicable. Une première solution consiste à utiliser les réseaux de Kohonen qui exploitent les données en entrée pour découvrir certaines régularités. La seconde s'appuie sur les algorithmes évolutionnaires pour calibrer les poids des arcs du réseau.

Réseaux de Kohonen

Les réseaux de Kohonen ou cartes auto-organisatrices (en anglais : Self Organizing Map, SOM) sont organisées autour d'un apprentissage non supervisé, et permettent de faire de la classification (ou clustering). Comme la règle de Hebb, les cartes SOM s'inspirent des processus biologiques : Les neurones de sorties sont mis en compétition et seul le neurone considéré comme le gagnant a la capacité de mettre à jour les poids des arcs qui lui sont attachés. Ce procédé se rapproche de la philosophie des algorithmes génétiques abordés au chapitre précédent.

Une variante possible consiste à choisir non pas un gagnant mais un groupe de gagnants pour lesquels la mise à jour des poids est effectuée. Ce groupe est choisi dans un voisinage du point gagnant. D'où la nécessité de définir une distance entre les neurones de sortie, ainsi qu'une topologie pour les problèmes à modéliser.

En pratique, les réseaux SOM sont constitués d'un ensemble de neurones d'entrée et d'un ensemble de neurones de sortie. Ce sont ces derniers qui vont être mis en compétition. Tous les neurones de sortie sont reliés à toutes les entrées. De manière plus formelle, le réseau peut être modélisé comme suit. Les valeurs d'entrée sont regroupées dans un vecteur $V = \{v_i,\ i = 1..n\}$, les neurones en compétition forment le vecteur $X = \{x_i,\ j = 1..m\}$ et les poids du réseau sont représentés par une matrice pleine (non-creuse) $P = \{p_{ij},\ i = 1..n,\ j = 1..m\}$.

Dans la matrice P, à chaque valeur de j correspond un vecteur de poids P_{*j} de même taille que le vecteur de données V. De plus, tous ces poids sont reliés au même neurone de sortie. Il est en quelque sorte considéré comme le représentant de ce neurone.

L'apprentissage se déroule comme suit :

Algorithme d'apprentissage :

1. Les valeurs de p_{ij} sont initialisées, et ce de manière aléatoire ou non.
2. Un vecteur d'entrées V est soumis au réseau. On détermine le $\widehat{\jmath}$ tel que la distance entre les vecteur V et P_{*j} est minimale. Le neurone $\widehat{\jmath}$ est considéré comme le vainqueur.
3. Le vecteur de poids $P_{*\widehat{\jmath}}$ est mis à jour de sorte que

$$P_{*\widehat{\jmath}} = P_{*\widehat{\jmath}} + \lambda(V - P_{*\widehat{\jmath}}).$$

   Ensuite, le scalaire λ est lui aussi modifié, sa valeur est diminuée selon une loi géométrique ou arithmétique.
4. S'il existe encore des entrées à soumettre au réseau, retour au deuxième point.

L'aspect géométrique du mécanisme d'apprentissage permet de mieux comprendre son fonctionnement. Les vecteurs de données V et de poids P_{*j} représentent des points de l'espace $\mathbb{R}^n$. Dans le but de faciliter les notations, les points qui représentent les vecteurs de données sont notés v_i, où i peut être arbitrairement grand en fonction du nombre d'apprentissages et les points qui représentent les vecteurs de poids P_{*j} sont notés p_j où $j \in \{1..m\}$.

L'apprentissage effectue le travail suivant. Pour chaque nouveau v_i considéré, le point p_j le plus proche est sélectionné et déplacé vers v_i. Si un autre point v_k est proche de v_i, ce même p_j sera une nouvelle fois le gagnant et sera déplacé dans la même direction. De cette manière, si les v_i ont une disposition tels qu'ils forment plusieurs nuages de points, et si le nombre et la disposition initiale des p_j n'est pas trop mal choisie, chacun des p_j va converger vers un des nuages de points. Au final, chaque p_j devient un représentant d'un nuage de points.

Une fois le réseau entrainé, chaque nouveau v_i est automatiquement classé dans un nuage de point et ce en fonction du p_j qui lui est le plus proche. Dans le réseau, les neurones de sortie servent à discriminer des ensembles distincts dans le jeu de données. Toutefois, le jeu de données doit présenter précisément cette organisation de sorte que le nombre de neurones de sortie coïncide.

Par conséquent, les cartes SOM permettent de faire de la classification de données. Malheureusement, ce type de classifications se range dans les méthodes de classifications les plus pauvres (cfr [4]).

Une carte SOM effectue un travail proche de celui effectué par le premier neurone présenté dans la section 1.2.3 à la page 17 où les données sont de types binaires et pour lesquelles l'utilisateur définit deux ensembles de données, l'un considéré comme positif et l'autre comme négatif. Par la suite, le neurone classe les nouvelles données en fonction d'une distance de Hamming.

Pour conclure, les réseaux de Kohonen, contrairement aux autres types de réseaux, n'exploitent pas les valeurs de sorties ni lors de l'apprentissage ni pendant son utilisation. Plus précisement, les neurones de sortie n'en sont pas vraiment car ils n'effectuent aucun calcul et ne jouent qu'un rôle symbolique, l'identification d'une classe.

Calibration par algorithme évolutionnaire

Une seconde méthode permettant de réaliser un apprentissage non supervisé d'un réseau de neurones consiste à employer un algorithme évolutionnaire. Les individus manipulés par celui-ci sont composés des poids du réseau dont les valeurs évoluent au fil des générations.

Etant donné que cet apprentissage ne se base pas sur une erreur par rapport à une solution souhaitée, aucune évaluation absolue de la valeur d'un candidat n'est disponible. Une comparaison des individus à l'aide d'une fonction *fitness* est nécessaire afin d'obtenir une évaluation relative des candidats et les classer selon un certain ordre. Une sélection est ensuite effectuée dans le but de générer la population suivante de l'algorithme évolutionnaire. Plusieurs recombinaisons et mutations peuvent être effectuées afin d'explorer au mieux l'espace des solutions.

L'arrêt de l'algorithme évolutionnaire peut être basé sur différents critères. Un premier consiste à fixer un nombre de générations maximal. Celui-ci a pour avantage de garantir l'arrêt de l'algorithme mais n'assure pas la qualité de la solution obtenue. Un second critère repose sur l'évolution de la population et force l'arrêt de l'algorithme lorsqu'une certaine stabilité est atteinte. L'inconvénient de cette méthode réside dans un temps d'exécution indéterminé qui peut être important, voire infini. Cependant, si une solution est retenue, celle-ci sera potentiellement de bonne qualité.

Chapitre 3

Etat de l'art

Afin de procéder à une analyse critique des techniques d'intelligence artificielle, nous procédons dans ce chapitre à un rapide tour d'horizon de différentes applications. Cette analyse nous permettra par la suite d'effectuer nos choix d'implémentation pour la réalisation d'un jeu de gestion multijoueurs.

3.1 Application à la robotique

Les voitures, les matériels électroménagers, ainsi que beaucoup d'autres outils de notre vie quotidienne utilisent l'informatique afin de simplifier leur utilisation par les êtres humains. Parmi les différentes fonctionnalités offertes, certaines ne nécessitent qu'une bonne interface entre l'homme et la machine afin d'être utilisées au mieux. Cependant, le futur de ce genre de technologies se trouve dans une intelligence propre à ce genre d'appareils.

De nombreux chercheurs travaillent sur des techniques d'intelligence artificielle capables de donner une *vie* à la machine afin qu'elle soit capable de prendre ses propres décisions. De nos jours, des aspirateurs automates sont commercialisés à grande échelle (cfr Figure 3.1). Cette machine est capable de se diriger par elle-même afin d'anticiper les collisions et de *nettoyer* toute une surface.

FIG. 3.1 – *Aspirateur autonome Roomba [59].*

Dans une expérience plus élaborée, des chercheurs ont montré qu'une voiture guidée par un réseau de neurones était capable de se diriger de manière autonome sur une route balisée (cfr Figure 3.2). Les pixels captés par une caméra située à l'avant de l'automobile constituaient les entrées du réseau de neurones. Ensuite, une sortie est générée par le réseau afin de produire un éventuel mouvement des roues.

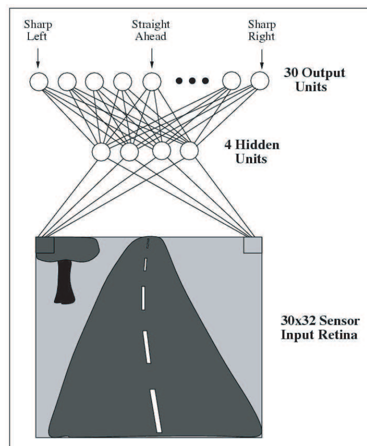

FIG. 3.2 – *Réseau de neurones lié au pilotage automatique d'un véhicule [2].*

Le challenge DARPA réunit plusieurs équipes dans le désert du Nevada afin d'y organiser une course de véhicules autonomes. En 2005, l'université de Stanford a remporté la victoire à l'aide de son véhicule *Stanley* doté d'un budget de $500,000.

L'élaboration d'un humanoïde constitue l'un des projets en vogue dans la recherche robotique. A ce jour, deux automates se détachent du lot. D'une part, *Hubo*, le fruit de 2 années de recherche d'une entreprise américaine, conçu à l'aide d'un budget d'un million de dollars. D'autre part, *Asimo*, le robot, plus ancien, de Honda dont la conception a nécessité 10 ans de recherche et pas moins de 300 millions de dollars (cfr Figure 3.3).

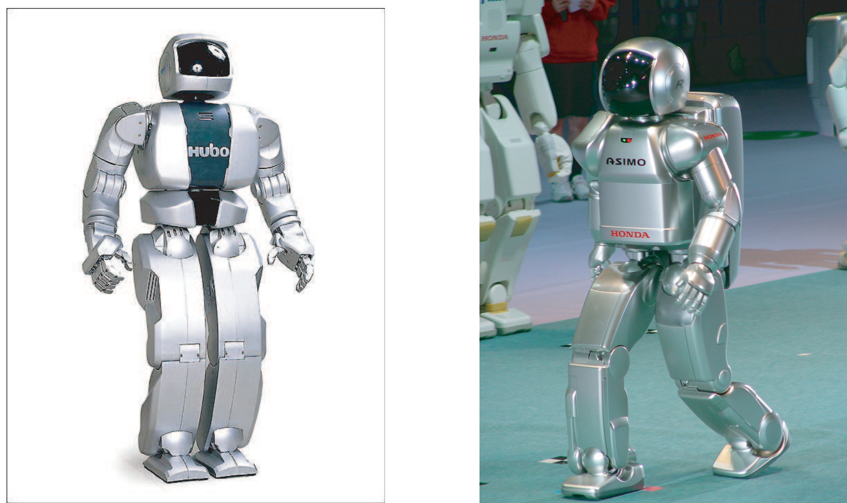

FIG. 3.3 – *Hubo et Asimo [59].*

Aujourd'hui, *Hubo* a encore évolué car les chercheurs lui ont greffé un visage humain, celui d'Albert Einstein. 31 moteurs permettent à l'humanoïde de reproduire les expressions faciales d'un être humain (rire, fermer ou cligner des yeux). La vidéo présentée par l'équipe de recherche est stupéfiante (voir [59]).

Le laboratoire *Animatlab* de Paris [49] effectue des recherches dans le but de développer des animaux artificiels capables de reproduire le plus fidèlement possible les mouvements de leurs homologues vivants (cfr Figure 3.4).

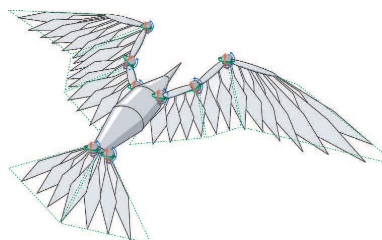

FIG. 3.4 – *Illustration de l'automate Robur [49].*

Le plus célèbre représentant de cette catégorie d'automates est le chien-robot *Aibo*. Développé par Sony, cet animal artificiel est le premier robot de compagnie commercialisé en masse. Doté de capteurs sensitifs, de commandes vocales ou encore d'un affichage LED pour exprimer certaines émotions, l'animal a déjà été vendu à plus de 150.000 exemplaires de part le monde. Cependant, l'entreprise nippone a annoncé, au début de cette année, l'arrêt de sa production d'*Aibo*. En effet, le coût en recherche et développement était trop important pour les profits engendrés.

FIG. 3.5 – *Aibo jouant au football [42].*

Une entreprise américaine spécialisée dans les technologies hybrides et l'énergie alternative a annoncé que l'*US Army* avait récemment commencé les tests de deux nouveaux véhicules autonomes de combat.

Nommés *Crusher*, ces deux véhicules autonomes alimentés d'électricité pèsent plus de 6 tonnes. Développé par une équipe d'ingénieurs dirigée par le National Robotics Engineering Center, cet énorme véhicule à 6 roues peut évoluer sur n'importe quel terrain sans jamais être gêné dans sa progression. Même renversé, il continue à rouler sur le dos sans aucune difficulté.

Ce prototype démontre à quel point l'armée américaine est à la pointe de la technologie. Il sera bientôt intégré aux troupes de l'*US Army* en compagnie d'autres innovations technologiques.

FIG. 3.6 – *"Crusher" [59].*

Une tortue robotisée pourrait très prochainement aider les ingénieurs à construire de meilleurs véhicules autonomes sous-marins et par la même occasion mettre en évidence la manière dont les animaux aquatiques nageaient pendant la préhistoire.

Baptisée Madeleine, cette tortue mesure 80 cm de longueur pour 30 cm de largeur et pèse 24 kg. Dotée de plusieurs capteurs dont un sonar, une caméra vidéo, une sonde de profondeur et un accéléromètre, la tortue robotisée imite le design de son homologue animal. En effet, celle-ci dispose également de nageoires car les chercheurs ont conclu que, si la nature en avait doté l'animal, c'est qu'il s'agissait de la meilleure solution à adopter pour réaliser les mêmes mouvements.

Cette découverte n'est peut être pas indispensable au monde de la robotique, mais elle jouera un rôle essentiel dans l'élaboration des futurs véhicules autonomes sous-marins.

FIG. 3.7 – *Tortue automate Madeleine [59].*

3.2 Application aux jeux vidéos

Depuis quelques années, le jeu vidéo investit le marché et s'avère être une source de profits économiques non-négligeable. Bon nombre de recherches dans le domaine de l'intelligence artificielle sont entreprises dans le but d'obtenir des ordinateurs capables de concurrencer l'être humain. La puissance des machines d'aujourd'hui alliée à des techniques d'intelligence artificielle de plus en plus développées permettent d'obtenir de tels résultats.

Etant donné que les cartes graphiques *dernière génération* sont capables de générer des images très proches de la réalité, l'avenir du jeu vidéo passe par le jeu multijoueurs sur internet. En effet, depuis quelques années, avec l'arrivée des lignes à haut débit, de nombreux jeux ont inclus une possibilité de jeu *online*. Actuellement, certains jeux se jouent uniquement sur l'Internet (*mmorpg*).

Cependant, dans ce genre de jeux, il est nécessaire d'avoir également des *personnages non-joueurs*, c'est-à-dire des ordinateurs capables de jouer et de se comporter comme des êtres humains. Les chercheurs s'attachent à développer des techniques d'intelligence artificielle de plus en plus perfectionnées afin que le joueur croit être confronté à des homologues humains.

FIG. 3.8 – *Jeux Black-and-White et Norns [21].*

Les réseaux de neurones représentent l'une des techniques les plus fréquemment utilisées dans le domaine du jeu vidéo multijoueurs. Nous avons pu voir que le réseau de neurones est capable d'apprendre automatiquement. Cependant, cet apprentissage nécessite un certain temps de calcul de la part de la machine et il risque de ralentir le jeu s'il est utilisé pendant la partie. L'ordinateur joue donc des parties *contre lui-même* afin que le réseau de neurones soit calibré et que le jeu soit efficace et cohérent. Le jeu *Black-and-White* de *Lionhead Studios* (cfr Figure 3.8) utilise cette technique. La créature contrôlée par l'ordinateur apprend à évaluer différents paramètres (ses motivations, son état et certains modificateurs) en fonction des actes du joueur adverse. Ensuite, en situation réelle, le système reproduit les actes qu'il a acquis dans des situations similaires. De nos jours, avec l'arrivée de cartes graphiques effectuant leurs propres calculs et libérant ainsi le processeur, de nouveaux jeux ayant une intelligence artificielle capable d'apprendre au cours de la partie sont développés. C'est le cas notamment de Norns qui

utilise des algorithmes génétiques au cours de la partie afin de procéder à une sélection naturelle et retenir les créatures s'adaptant le mieux à l'univers créé par le joueur (cfr Figure 3.8).

3.3 Application à l'optimisation mathématique

Comme nous l'avons déjà mentionné brièvement lors de la présentation des algorithmes évolutionnaires, ceux-ci offrent une méthode stochastique d'optimisation globale en mathématique.

Si nous considérons un problème de la forme (3.1), où f, g et h sont des fonctions non-linéaires, les algorithmes mathématiques courants n'y apportent souvent qu'une solution locale (LANCELOT, KNITRO [60]).

$$\begin{array}{lll} \min & f(x) & \\ \text{s.c.} & g_i(x) \leq 0 & i = 1, \ldots, m \\ & h_j(x) = 0 & j = 1, \ldots, p \end{array} \qquad (3.1)$$

où $x \in \mathbb{R}^n$.

Le principal avantage des algoritmes évolutionnaires réside dans le fait qu'ils permettent d'explorer l'espace de recherche de façon plus complète afin de fournir une solution globale au problème. Ils sont également capables de trouver une solution à un problème d'optimisation dont la fonction objectif f ne possède pas de structure particulière (non-continue par exemple).

3.4 Prévisions cinématographiques

Dans récent article [25], Ramesh Sharda et Dursun Delen de l'Oklahoma State University ont développé un réseau de neurones qui tente de prédire le succès financier d'un film avant sa sortie en salle. Ce problème est particulièrement difficile à résoudre puisque cette prédiction doit être faite avant même que les premiers spectateurs ne puissent voir le film. D'autres approches utilisent les chiffres d'affluence de la première journée ou de la première semaine de sortie qui permettent de préciser les résultats. Ces données permettent ainsi d'obtenir plus facilement un meilleur indice de prédiction. En effet, 25% des revenus d'un film sont générés durant les deux premières semaines

de sortie il est donc plus aisé de prédire son succès en utilisant ces chiffres.

Malgré le défi qu'elle représente, la prédiction *avant sortie* a déjà fait l'objet par le passé d'autres études (essentiellement des études statistiques). Celles-ci représentent une bonne base de comparaison. L'utilisation des réseaux de neurones constitue une première dans ce domaine. L'objectif de cette application est de classer le film dans une catégorie qualifiant son succès. Ces différentes catégories sont une discrétisation du montant des recettes générées . Elles sont au nombre de neuf et débutent par la classe *moins d'un million de dollars* (flop) pour se terminer par la classe *plus de 200 millions de dollars* (blockbuster).

La discrétisation des données offre de nombreux avantages. En effet, une fois simplifiées, les données sont plus faciles à manipuler, à comprendre ou à interpréter. Cette technique rend la plupart des algorithmes d'apprentissage plus rapides et plus précis.

Les variables en entrée du problème sont au nombre de sept et ont été choisies en fonction d'études précédentes (voir Figure 3.9) :

- **La classification MPAA** (Motion Picture Association of America), qui est l'équivalente du CSA (Conseil supérieur de l'audiovisuel). Celle-ci classe tous les films en fonction de plusieurs critères comme la présence de scènes violentes ou de dialogues trop rudes.

- **La concurrence des autres films**. Par rapport à sa date de sortie, un film entre en compétition avec un nombre plus ou moins grand de concurrents.

- **La popularité de la tête d'affiche**. Cette variable permet de quantifier la notoriété des acteurs. Elle est calculée en fonction des derniers revenus des acteurs.

- **Le genre du film**. Il peut être classé dans une ou plusieurs catégories telles que comédie, drame ou science-fiction.

- **Les effets spéciaux**. La variable déterminée par les effets spéciaux est calquée sur une classification faite par un site web cinématographique

de référence (`www.showbizdata.com`).

- **La suite**. Ce facteur caractérise un film faisant l'objet d'une suite. non.

- **Le tirage**, c'est-à-dire le nombre d'écrans sur lesquels le film sera projeté simultanément.

A partir de ces sept variables, Sharda et Delen ont choisi de les traduire en vingt-six pseudo-variables binaires. Par exemple, la variable "popularité de la tête d'affiche" est transformée en un triplet binaire. Le premier bit sera non-nul pour un film sans vedette, tandis qu'un film avec une grosse affiche sera représenté par un triplet 001. Seule la variable représentant le tirage échappe à cette transformation.

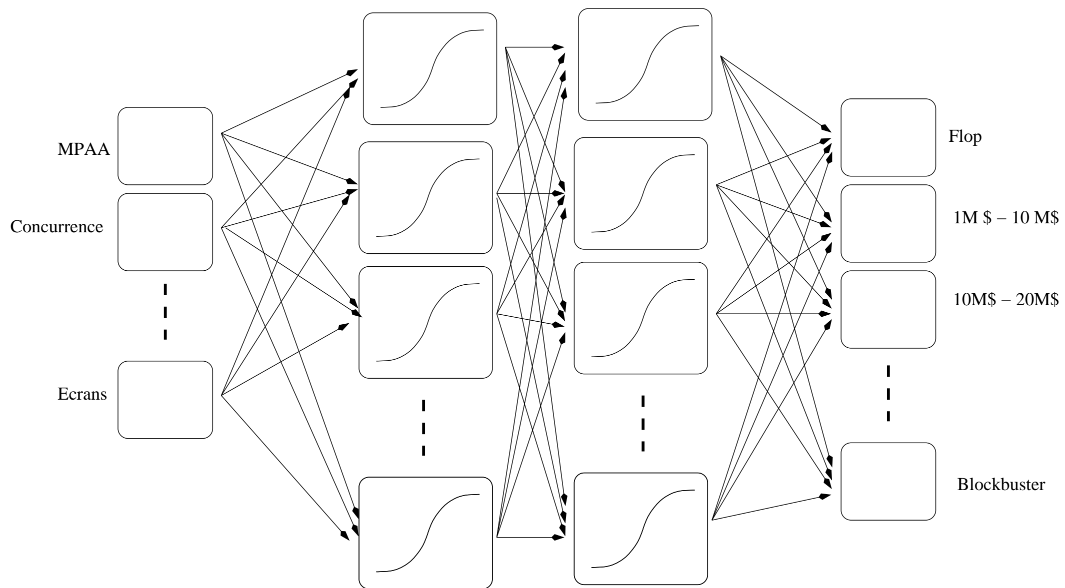

FIG. 3.9 – *Structure du réseau de neurones utilisé par Ramesh Sharda et Dursun Delen.*

Afin de comparer objectivement différentes structures de réseaux, les auteurs ont mis en place un système de validation croisée appellé *k-fold*. Pour réaliser cette validation, l'ensemble des données utilisées pour cette étude est composé des résultats des 834 films (sortis entre 1998 et 2002 inclus). Cet

ensemble est ensuite découpé de différentes manières en deux groupes : un groupe d'apprentissage et un groupe de tests. La validation croisée consiste à évaluer le réseau sur toutes les combinaisons de découpage. Cette méthode permet d'apporter de la rigueur quant à la manière de comparer les candidats.

Suite aux tests effectués sur cet ensemble de films, la structure retenue est composée de 26 neurones d'entrée, 9 neurones de sortie et 2 couches cachées. Celles-ci contiennent 18 et 16 neurones dont les fonctions de transfert sont des sigmoïdes (cfr Figure 3.9).

Les résultats du modèle sont mesurés vis-à-vis de deux métriques. La première se fonde sur le pourcentage de films pour lesquels la prédiction est correcte, c'est-à-dire le pourcentage de films qui ont été placés dans la bonne catégorie par le réseau (les auteurs parlent dans ce cas de *bingo*). La seconde est moins stricte, puisqu'elle est basée sur le pourcentage de films pour lesquels le modèle s'est trompé d'au plus une catégorie (cette situation est appelée *one-away*).

Le réseau de neurones prédit correctement en moyenne 36.9 % des films (*bingo*) et 75.2 % des films sont dans une situation *one-away*.

L'article [25] reprend les résultats de trois méthodes statistiques pour offrir une base de comparaison :

- La régression logistique dont les résultats sont de 30.17 % et 69.6%.
- L'analyse discriminante dont les résultats sont de 29.25 % et 67.69%.
- La classification (basée sur un arbre de régression) dont les résultats sont de 31.18 % et 71.07%.

L'approche utilisée par Ramesh Sharda et Dursun Delen obtient des résultats satisfaisants. L'utilisation de réseaux de neurones offre donc une bonne alternative aux méthodes plus classiques quand le problème à résoudre est particulièrement complexe.

Si les chiffres de la première semaine de vente sont pris en compte pour la prédiction, l'avantage de l'utilisation d'un réseau de neurones devient moins net. Un autre inconvénient de l'emploi de cette technique réside dans le

fait que le réseau fonctionne comme une boîte noire. Ainsi, les méthodes statistiques traditionnelles sont nécessaires pour pouvoir dégager les facteurs qui influencent le succès d'un film.

3.5 Prévisions boursières

Christian Dunis et Mark Williams ont publié en 2002 un article intitulé *"Modelling and Trading the EUR/USD Exchange Rate: Do Neural Network Models Perform Better?"* [24]. Cet article explique leur tentative d'utiliser un réseau de neurones pour prévoir les variations du taux de change entre l'Euro et le Dollar.

Ce taux d'échange entre les deux devises dépend de nombreux facteurs. En effet, ceux-ci peuvent être de type économique (croissance, taux d'intérêt ou inflation des pays respectifs), psychologique (échanges spéculatifs) ou politique. Les interactions entre ces différents facteurs sont complexes et rendent les prévisions difficiles.

L'historique des prix d'échanges (également appelé séquence) fait également partie de ces facteurs. Ces séries posent un premier problème car les mouvements des taux de change sont gouvernés par des tendances à long terme et sont généralement non stationnaires. En d'autres termes, elles ne sont pas stables autour de leurs moyennes. Or la stabilité d'une série est une propriété statistique utile et souhaitable. Pour obtenir des séries stationnaires, une transformation est appliquée aux taux de change. Cette modification définit le *taux de rendement* qui est basé sur la variation du taux entre deux périodes:

$$R_i = \frac{T_i}{T_{i-1}} - 1$$

où T_i est le taux d'échange au temps i.

Les séries de prix posent en général un autre problème. En effet, si leurs comportements respectent approximativement une tendance à long terme, celle-ci est généralement aléatoire à court terme, pénalisant ainsi les mécanismes de prédictions. Pour éviter cet inconvénient, un changement d'échelle peut être appliqué. Au lieu de considérer la valeur du cours de transaction à chaque heure, seuls les prix en fin de journée sont conservés. En revanche, ce changement d'échelle est inapplicable pour des prévisions à court terme.

Les autres données utilisées dans les prévisions sont composées de 19 indices boursiers comme le CAC40 de la bourse de Paris, l'indice *footsie* de Londres ou encore le cours du baril de Brent. Ces données sont choisies à l'aide d'une analyse en composantes principales qui fournit une manière de décaler les données dans le temps. En effet, les échanges *Euro-Dollar* s'effectuent 24h/24 7j/7[1] dans le monde entier. Les informations provenant des différentes bourses mondiales ont un impact qui varie dans le temps suivant le pays (Bruxelles ou Tokyo par exemple). Il est donc nécessaire de considérer un décalage dans le temps (*lag*) entre les différents indices boursiers et de choisir pour chacun d'eux la valeur de *lag* la plus significative.

Dans cet article, les auteurs motivent l'utilisation d'un réseau de neurones par deux raisons. Tout d'abord, les RNA sont capables de modéliser des relations fortement non-linéaires entre les variables indépendantes et les variables dépendantes (les prévisions) ce qui n'est pas le cas avec les modèles traditionnels dans le domaine boursier. De plus, la méthode à utiliser pour modéliser ce type de problèmes est souvent difficile à identifier, il est donc préférable de s'appuyer sur un réseau de neurones qui est une généralisation de ces méthodes.

Pour évaluer correctement les performances du RNA, le jeu de données est découpé en trois groupes : un ensemble d'entraînement, un ensemble de test et un ensemble de validation pour former une partition 2/3 - 1/6 - 1/6.

L'ensemble de test est utilisé pour connaitre l'évolution de la qualité du réseau au cours de l'apprentissage. Il permet de stopper celui-ci lorsque les résultats sont satisfaisants. En effet, un apprentissage prolongé rendrait le réseau *sur-adapté* et augmenterait le nombre d'erreurs sur le reste des données. Une fois l'entraînement correctement effectué, la performance du RNA est calculée en utilisant des données non-utilisées lors de l'apprentissage, à savoir les données de l'ensemble de validation.

Le réseau de neurones utilisé est classique puisqu'il est composé de neurones standards (de McCulloch et Pitts) avec une fonction d'activation en sigmoïde. Le nombre de couches est déterminé à l'aide d'une méthode empirique. Un réseau est initialisé avec une seule couche cachée contenant peu de neurones et se complexifie progressivement. Chaque version est ensuite évaluée et la meilleure d'entre elles est retenue. Cette manière de procéder est

1. 24 heures sur 24, 7 jours sur 7

nécessaire pour trouver un bon compromis entre un RNA trop complexe, qui pénalise le temps d'apprentissage, et un RNA trop simple, qui est incapable de s'adapter correctement aux données.

Le réseau est entrainé pour évaluer un taux de rendement, celui-ci est positif si le taux de change augmente, nul s'il est stable et négatif sinon. De plus, son amplitude est proportionnelle à l'importance de la variation du taux de change. Les deux métriques statistiques utilisées pour évaluer les performances du modèle sont l'erreur absolue moyenne du calcul du taux de rendement et le pourcentage de prévisions pour lesquelles le signe de la tendance est correctement prédit [2].

En comparant les différents modèles de prévision économiques et statistiques suivant les deux métriques définies ci-dessus, nous remarquons que le RNA offre les résultats les plus satisfaisants. Cependant, malgré ses bons résultats, les performances du réseau de neurones sont encore insuffisantes pour que celui-ci soit utilisé en tant qu'appliquation industrielle. En effet, pour atteindre la rentabilité, un bureau de courtage doit dépasser un pourcentage de décisions de vente ou d'achat *gagnantes* supérieur à 60%. Or, en utilisant le RNA, ce pourcentage plafonne à 57%.

3.6 Discussion

Lors de cette présentation de diverses applications liées à l'intelligence artificielle, nous avons pu constater que les réseaux de neurones offraient certains avantages. En effet, ils permettent d'aborder de nombreux problèmes complexes sans en avoir une connaissance approfondie. Cependant, une analyse a posteriori n'est pas toujours convaincante car le système agit plutôt comme une boîte noire.

Quant à l'utilisation d'algorithmes évolutionnaires, elle permet d'obtenir un ensemble de solutions et de choisir parmi celles-ci la plus appropriée au problème posé. De plus, l'évolution engendrée par l'algorithme donne une réelle impression de vie dans un univers créé par l'application, qu'il soit de nature mathématique ou ludique.

2. Une tendance à la hausse correspondant à un taux de rendement positif.

Chapitre 4

Etude de cas

Afin d'implémenter un moteur d'intelligence artificielle pour un jeu de gestion multijoueurs, nous avons réfléchi à une manière d'appliquer les différentes techniques que nous avons présentées dans les chapitres précédents à ce type de jeux. L'étude de différentes applications nous a permis de constater quelles sont les plus utilisées et celles qui seraient le mieux adaptées à notre système.

Dans ce chapitre, nous présentons le jeu sur lequel nous avons décidé de faire quelques expérimentations numériques, ainsi que l'adaptation de différentes techniques d'intelligence artificielle à notre problème.

4.1 Présentation du jeu

Le jeu que nous avons décidé d'implémenter se nomme *Korsar* aux éditions *Tilsit.* Il s'agit d'un jeu de gestion multijoueurs (de 3 à 8 joueurs) composé de cartes de deux natures différentes. D'un côté, nous trouvons les cartes *Galion* qui représentent des pièces d'or à gagner. De l'autre côté, les cartes *Pirate* qui représentent des points d'attaque (voir Figure 4.1). Le but du jeu est de collecter un maximum de pièces d'or en attaquant les cartes *Galion* à l'aide de cartes *Pirate.* Le vainqueur est le joueur ayant gagné le plus de pièces d'or à la fin de la partie.

FIG. 4.1 – *Cartes de jeu Galion et Pirate.*

4.1.1 Contenu du jeu

Le jeu est composé :

- de 25 cartes *Galion* dont la valeur va de 2 à 8.
- de 48 cartes *Pirate*, 12 cartes dans chacune des 4 couleurs (rouge, bleu, verte et jaune), dont la valeur va de 1 à 4.

4.1.2 Mise en place

Au début de partie, chaque joueur reçoit 6 cartes du talon de 73 cartes mélangé au préalable. Le talon est ensuite placé au centre de la table, il constituera la pioche.

4.1.3 Déroulement du jeu

Le tour de jeu d'un joueur est composé de deux opérations successives :
- Tout d'abord, s'emparer des éventuelles cartes *Galions* contrôlées
- Ensuite, jouer ou piocher une carte.

4.1.4 Piocher, jouer ou défausser

Lors de son tour de jeu, un joueur a le choix entre l'une des 2 opérations suivantes :
- Piocher une carte dans le talon
- Jouer une carte sur la table

Une fois vide, le talon n'est pas renouvelable. Ainsi, l'opération de *piocher* est limitée. Nous verrons plus tard que la fin de partie est proche.

4.1.5 Cartes

Lorsqu'un joueur pose une carte sur la table, l'opération diffère en fonction de la nature de la carte jouée.

Cartes Galion

Une carte *Galion* doit être posée face visible devant le joueur l'ayant jouée. Elle constitue une quantité de points à gagner.

Cartes Pirates

Les cartes *Pirate* permettent d'attaquer ou défendre les cartes *Galion* placées sur la table. Lorsqu'un joueur désire attaquer ou défendre un galion, il place une carte pirate face à lui et dirigée vers le galion en question. Si le tour suivant, le joueur désire encore attaquer ou défendre ce galion, il devra obligatoirement le faire dans la couleur qu'il a déjà jouée. Si plusieurs joueurs attaquent un même galion, ils doivent nécessairement le faire chacun avec une couleur différente. Typiquement, les cartes sur la table seront disposées comme présenté à la figure 4.2.

4.1.6 S'emparer des cartes Galion

Au début de son tour de jeu, un joueur gagne les cartes *Galion* qu'il contrôle. La règle est simple, il contrôle le galion s'il possède plus de cartes

FIG. 4.2 – *Disposition des cartes pirates.*

pirates (en force et non en nombre de cartes) que tout autre joueur. La carte *Galion* gagnée est placée face cachée à côté du joueur. Les cartes *Pirate* anciennement placées sur ce galion sont défaussées. Elles ne feront plus partie du jeu par la suite. Si aucun pirate n'a été posé sur un galion que le joueur a posé le tour précédent, celui-ci est gagné et est placé dans son butin.

4.1.7 Fin de la partie & Décompte des points

Lorsque la dernière carte du talon est piochée, la partie continue jusqu'à ce que l'un des joueurs *joue* sa dernière carte. A partir du moment où le talon est vide, les joueurs ont la possibilité de défausser une carte lors de leur tour de jeu au lieu d'en jouer une. Cependant, la carte défaussée ne peut pas être une carte *Galion*. En effet, ces cartes doivent faire partie du décompte des points. A la fin de la partie, les galions se trouvant encore sur la table sont gagnés par les joueurs qui les contrôlent à ce moment-là. Les galions attaqués par des pirates de force égale sont laissés sur la table et n'interviendront pas dans le décompte des points. Chaque joueur fait ensuite la somme des cartes *Galion* qu'il a gagnées au cours de la partie. Il retire ensuite de cette somme les éventuels galions qu'il possède encore dans sa main. Le vainqueur est le joueur ayant le plus gros butin, il débutera la prochaine partie.

4.2 Intelligence artificielle

Afin de mettre en place une intelligence artificielle pour ce jeu, nous avons tout d'abord dû nous familiariser avec celui-ci. Nous avons donc joué plusieurs parties afin d'établir certaines stratégies et de pouvoir analyser par la suite si le moteur que nous allions mettre en place jouerait de façon cohérente. Nous avons décidé d'implémenter deux types d'intelligences artificielles : l'une utilisant les algorithmes évolutionnaires pour calibrer un arbre de décision probabiliste, l'autre utilisant un réseau de neurones calibré au préalabre par un algorithme évolutionnaire également (apprentissage non-supervisé).

4.2.1 Algorithmes évolutionnaires

La première étape dans l'implémentation d'un algorithme évolutionnaire est l'encodage des données sous forme binaire. Dans le cas d'un jeu multijoueurs, il faut procéder à une automatisation d'un raisonnement de joueur. Nous avons donc introduit divers paramètres qui fixent le comportement d'un joueur dans des situations précises. Par exemple, à tout moment de la partie, lorsque le talon est non-vide, le joueur a la possibilité de piocher ou de jouer une carte. Afin de respecter les règles du jeu, nous avons dû également implémenter directement certains comportements. En effet, lorsqu'aucun galion n'est sur la table et que le joueur ne possède que des cartes pirates, la pioche est inévitable.

Nous avons donc associé à chaque joueur une séquence binaire qui représente son comportement dans ces situations. Afin d'appliquer l'algorithme évolutionnaire, nous avons généré une population initiale exhaustive en générant toutes les combinaisons possibles de séquences binaires pour l'ensemble des paramètres.

Notre évaluation des joueurs est basée sur leur aptitude à gagner des parties de jeu. Etant donné que la distribution des cartes influence le résultat lors d'une partie avec des joueurs de même niveau, et que celle-ci s'effectue de manière aléatoire, nous devions procéder à un ensemble de parties (1000) pour déterminer le joueur ayant un pourcentage de victoires plus important que les autres. Un des défauts de cette technique réside dans le fait que nous ne pouvons comparer que quatre joueurs afin d'en sélectionner un pour la génération de la population suivante.

Afin de disposer d'une population de taille équivalente, nous devions générer 3 nouveaux joueurs à partir du meilleur sélectionné. Nous procédions donc à plusieurs recombinaisons, ainsi qu'à une éventuelle mutation, pour obtenir les individus souhaités.

En effectuant plusieurs itérations de notre algorithme évolutionnaire, nous disposions d'une population de joueurs capables de jouer de façon cohérente. Cependant, après avoir analysé leur façon de jouer en se confrontant à eux plusieurs fois, il était relativement aisé de prédire leur comportement.

Suite à cette constatation, nous voulions obtenir un comportement *plus humain* de nos joueurs afin de pouvoir jouer contre ceux-ci sans pouvoir anticiper leurs décisions. Cependant, il fallait respecter une certaine cohérence dans les actions des joueurs. Nous avons donc opté pour l'introduction de probabilités au sein de notre modèle. Lors de notre encodage, nous avons défini chaque paramètre comme étant une action en fonction d'une situation précise. Cette action était de forme binaire (jouer ou piocher par exemple). L'introduction de probabilités permet de nuancer ce choix et d'éviter les comportements déterministes dans des situations similaires.

Nos opérateurs de mutation ont dû être modifiés afin de respecter les modifications des algorithmes évolutionnaires dans le cas de données réelles. Cependant, nous avons veillé à ce que nos paramètres restent compris entre 0 et 1.

Suite à cette modification, nous avons de nouveau appliqué notre algorithme à notre problème. Une certaine convergence des paramètres était également visible. En effet, parmi ceux-ci, nous pouvions constater que les joueurs ayant une probabilité très faible d'attaquer les galions mis sur la table plutôt que de piocher voyaient leurs représentants dans la population diminuer au détriment des joueurs *agressifs*. Par contre, d'autres paramètres, tels que le choix du galion à attaquer ou le choix de la carte pirate à placer sur ce galion, ne convergeaient pas vers une valeur précise. Nous avons donc conclu que ce paramètre importait peu dans le jeu du joueur.

En jouant contre cette nouvelle génération de joueurs, nous avons pu constater que leur comportement était beaucoup moins prévisible. Cependant, certaines situations avaient toujours lieu. En effet, en début de chaque partie, les joueurs étaient réticents à placer un galion sur la table par crainte d'être attaqué. Par contre, une fois que la première carte Galion était placée,

les joueurs attaquaient ou posaient des galions de façon cohérente. Le comportement de ces joueurs se rapprochant plus du comportement humain, il était beaucoup plus difficile de prévoir leurs coups.

Il est à noter que le jeu du joueur artificiel dépend également de la façon dont les paramètres sont utilisés et de la précision de la représentation de la situation de jeu. Afin d'obtenir une vision plus précise de celle-ci, nous étudions par la suite l'application des réseaux de neurones à ce genre de problèmes.

Aspects techniques & Résultats

Dans cette section, nous précisons certains détails quant aux stratégies implémentées et aux résultats obtenus.

Premièrement, nous avons établi les opérations *génétiques* telles que la recombinaison ou la mutation. Ainsi, la mutation d'un individu est basée sur une perturbation de ses différentes caractéristiques, i.e.

$$mutant_i = genome_i + (-1)^r.s.paramMutation$$

où r est un entier aléatoire entre 0 et 1, s un réel aléatoire entre 0 et 1 et $paramMutation$ est choisi afin d'obtenir une certaine convergence de l'algorithme (typiquement, de l'ordre de 0.01).

Quant à la recombinaison, elle est obtenue à l'aide d'un *cross-over* de type intermédiaire, i.e.

$$Enfant = (1 - \alpha)parent_1 + \alpha parent_2$$

où α est un réel aléatoire choisi entre 0 et 1.

Ces deux choix d'opérateurs ont été effectués dans des buts bien précis. Premièrement, étant donné que nous utilisons des probabilités, nous avons privilégié l'adoption d'une mutation plus régulière mais de faible amplitude. Deuxièmement, la recombinaison choisie permet d'explorer de manière plus approfondie l'espace de recherche. En effet, la population initiale que nous générons se trouve sur la périphérie de cet espace et une recombinaison de type standard ne permet pas de parcourir l'entièreté du domaine.

Ensuite, les choix du joueur sont effectués en fonction de l'état du jeu. Par exemple, lorsque le talon est non-vide, plusieurs possibilités sont envisageables. Tout d'abord, la table peut être vide, c'est-à-dire qu'aucun galion n'a encore été posé par un joueur. Dans ce cas, l'ordinateur a éventuellement le choix entre 2 actions : piocher ou jouer un galion en mer (s'il en possède un en main). Un paramètre ($Param[1]$) de notre intelligence artificielle est basé sur ce choix et les comportements suivants sont obtenus en fonction d'un nombre aléatoire généré par l'application

$$\begin{cases} r \leq Param[1] \rightarrow \text{Piocher} \\ r > Param[1] \rightarrow \text{Jouer un galion} \end{cases}$$

où r est choisi de manière aléatoire entre 0 et 1.

Une autre situation de jeu est envisageable quand un joueur ne dispose pas de cartes galions en main mais a la possibilité d'attaquer ceux se trouvant sur la table [1]. Dans ce cas, le paramètre correspondant à cette situation est utilisé de la manière suivante

$$\begin{cases} r \leq Param[0] \rightarrow \text{Piocher} \\ r > Param[0] \rightarrow \text{Attaquer un galion en mer} \end{cases}$$

où r est choisi de manière aléatoire entre 0 et 1.

Deux autres paramètres permettent de choisir l'action à effectuer lorsque les 3 actions sont envisageables (attaquer, mettre un galion sur la table ou piocher). Nous avons combiné deux tests basés sur des nombres aléatoires comme précédemment.

Le schéma 4.3 présente de manière symbolique la façon dont sont exploités les paramètres dans la prise de décision de l'intelligence artificielle. Comme nous pouvons le constater, certains choix sont basés sur des probabilités, tandis que d'autres sont déterminés par la situation de jeu dans laquelle nous nous trouvons. Sur cette figure, les termes GST et GEM signifient respectivement que des galions se trouvent sur la table et que le joueur en possède dans sa main. Lorsque l'un de ceux-ci est barré, cela signifie que nous nous trouvons dans la situation opposée.

1. Une couleur pirate disponible en main est libre sur un galion

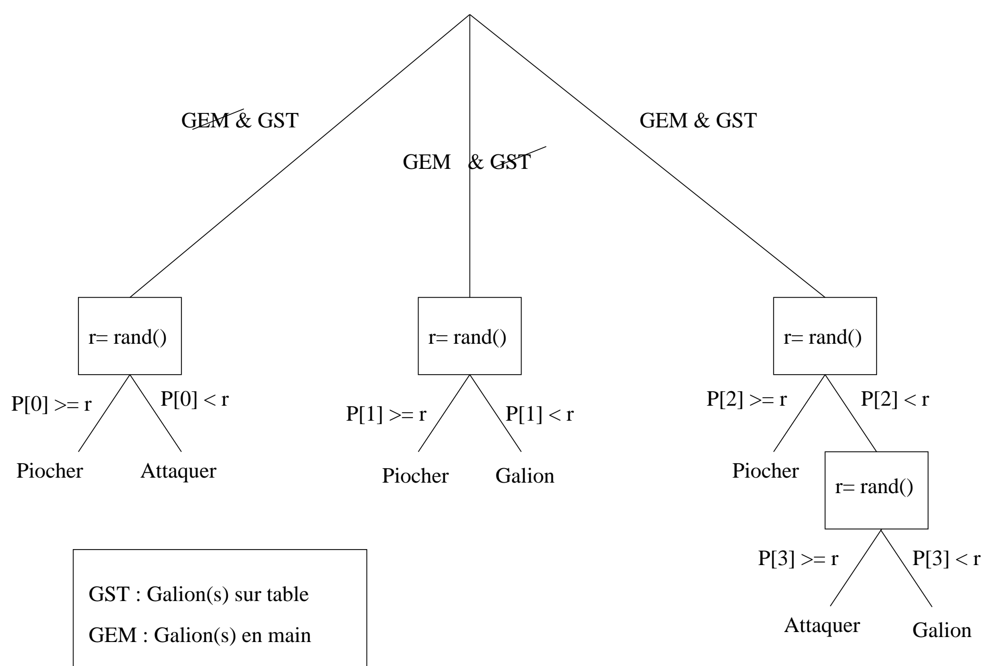

FIG. 4.3 – *Représentation symbolique de l'intelligence artificielle.*

Pour résumer les résultats obtenus, nous analysons la moyenne des paramètres sur l'ensemble des individus. Des valeurs extrêmes (proches de 0 ou proches de 1) attestent d'une uniformité des valeurs de la population. Suite à cette première implémentation, nous avons obtenu les résultats suivants

Moyenne du paramètre 0	0.148
Moyenne du paramètre 1	0.820
Moyenne du paramètre 2	0.179
Moyenne du paramètre 3	0.821

TAB. 4.1 – *Premiers résultats.*

Ces résultats illustrent le fait qu'une plus grande proportion d'individus ayant un *génome* proche de

$$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

est présente dans la population finale. Cela signifie que les joueurs adoptant plus régulièrement la stratégie associée à ces valeurs obtiennent de meilleurs résultats. De tels joueurs vont favoriser l'attaque à la pioche ($p[0] \cong 0$) quand ils n'ont pas de galions en main et que des galions se trouvent sur la table. Par contre, ils vont privilégier le fait de piocher lorsqu'ils possèdent des galions et qu'aucune carte de ce type ne se trouve sur la table ($p[1] \cong 1$). Enfin, dans le cas où des galions sont disponibles en main et présents sur la table, ils vont considérer par ordre de préférence, les actions d'attaquer, jouer un galion et piocher ($p[2] \cong 0$ et $p[3] \cong 1$).

A l'aide de ces paramètres, nous obtenons des automates jouant de façon cohérente. Cependant, nous voulions raffiner leurs comportements en introduisant de nouveaux points de décision basés sur le choix du galion à attaquer ou à défendre. La signification des différents paramètres ajoutés est présentée dans le tableau 4.2.

$param[4]$	proba de choisir la défense de son propre bateau
$param[5]$	proba d'attaquer le plus gros galion
$param[6]$	proba d'attaquer le plus petit galion
$param[7]$	proba de défendre le plus gros bateau mis en mer
$param[8]$	proba de défendre le plus petit bateau mis en mer

TAB. 4.2 – *Signification des paramètres supplémentaires.*

La figure 4.4 présente de façon schématique le choix du galion à attaquer. Les différentes prises de décision permettent de sélectionner le galion le plus fort, le plus faible ou aléatoirement.

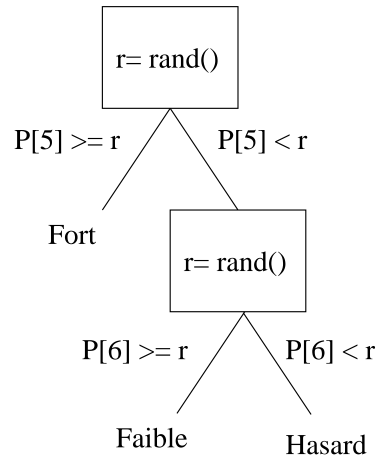

FIG. 4.4 – *Choix du galion.*

En ce qui concerne le choix de la carte pirate, nous n'avons pas introduit de paramètres particuliers. Par contre, le joueur artificiel évite de poser une carte pirate sur un galion s'il n'arrive pas à renchérir suffisamment pour en prendre possession.

Suite à l'application de notre algorithme évolutionnaire sur une population composée de ces différents paramètres, nous avons obtenu des résultats illustrés dans le tableau 4.3.

Moyenne du paramètre 0	0.179
Moyenne du paramètre 1	0.914
Moyenne du paramètre 2	0.179
Moyenne du paramètre 3	0.866
Moyenne du paramètre 4	0.570
Moyenne du paramètre 5	0.448
Moyenne du paramètre 6	0.632
Moyenne du paramètre 7	0.388
Moyenne du paramètre 8	0.481

TAB. 4.3 – *Résultats obtenus.*

Comme nous pouvons le constater, certains paramètres n'ont pas de ten-

dance particulière (0.6 , 0.4 , ...). En analysant de façon plus détaillée la population résultante, nous avons constaté une forte variance entre les différents individus. En effet, il existe une proportion pratiquement équivalente de joueurs où le paramètre est élevé que de joueurs où celui-ci est faible. Dans un tel cas, nous considérons que le paramètre n'est pas trop important pour assurer la qualité du joueur.

Suite à cette analyse, nous avons décidé de générer un joueur artificiel de référence en initialisant chacun de ses paramètres à la moyenne de celui-ci sur l'ensemble de la population. L'intelligence artificielle du joueur obtenu permettra ainsi de se refléter à un comportement humain (choix moins déterministe).

En jouant des parties manuellement face à ce joueur artificiel, nous avons constaté que les résultats étaient satisfaisants. En effet, l'automate atteint le niveau d'un joueur moyen et n'effectue pas toujours les mêmes opérations dans des situations relativement similaires (non-déterminisme). De plus, par la nature de l'approche utilisée, nous avons la possibilité d'analyser son comportement, ce qui s'avère très utile pour le développement d'autres techniques.

Cependant, cette manière d'analyser les résultats ne permet pas de considérer le couplage de différents paramètres. C'est pourquoi, nous avons tenté une approche orientée *Réseaux de neurones* pour ce type de problèmes dans la suite de nos expérimentations.

4.2.2 Réseaux neuronaux

Notre première réalisation a été conçue dans le but de tester faisabilité d'une telle idée. Le problème se prête bien à l'utilisation d'un réseau de neurones. En effet, il est délicat de mettre en place un algorithme traditionnel performant pour un jeu où l'influence mutuelle des joueurs affecte autant les prises de décisions.

La modélisation complète de l'intelligence artificielle du jeu *Korsar* en utilisant uniquement un réseau de neurones nous paraît pratiquement impossible. De par sa nature, un jeu de cartes impose des prises de décisions bien plus complexes qu'une simple alternative binaire. En particulier, pour *Korsar*, l'action de jouer passe d'abord par un choix entre les trois types

de comportements principaux : attaquer, défendre ou piocher. Pour deux de ces choix (attaquer et défendre), une autre décision doit encore être prise. En effet, lors d'un coup d'attaque, le joueur doit choisir le couple de cartes *galion/pirate* qui sera le plus efficace en fonction des cartes disposées sur la table et des cartes disponibles en main. Dans ce cas, le nombre de coups possibles est proportionnel au produit du nombre de galions sur la table et du nombre de pirates en main.

Pour simplifier cette situation, nous avons choisi de limiter le rôle du réseau de neurones et de l'utiliser uniquement pour déterminer le type de coups parmi les trois à effectuer. Ensuite, le choix de la carte à jouer est implémenté via des règles plus simples. Par exemple, pour un coup d'attaque, l'intelligence artificielle attaquera systématiquement le galion le plus important avec son pirate le plus puissant.

Structure du réseau

La première étape de la création du RNA consiste à choisir les données à fournir aux neurones d'entrée. Ce choix va dépendre d'un compromis. En effet, une première couche trop importante augmente le nombre d'arcs et la taille du réseau. Cela peut s'avérer inefficace et surtout nuire à la convergence de l'apprentissage. Par contre, si les valeurs données en entrée sont trop pauvres, le manque d'informations risque de compromettre la qualité des résultats.

Finalement, nous avons choisi d'adopter une couche d'entrée composée de 96 neurones. Le premier neurone comptabilise le nombre de cartes déjà piochées dans le talon et les 23 suivants représentent la main du joueur. Ces derniers jouent un rôle bien précis. Parmi ceux-ci, les 7 premiers comptabilisent le nombre de galions correspondants, de sorte que le $i^{\text{ème}}$ neurone contienne le nombre de galions de valeur i. Les 16 autres neurones restants additionnent le nombre de cartes pirate de chaque type. Le type dépend de la couleur (rouge, vert, jaune ou bleu) et de la force (de une à quatre étoiles).

Le suite des neurones d'entrée modélise l'état de la table. Pour simplifier la quantité de données à manipuler, le réseau considère qu'un joueur ne pose au maximum que 3 galions simultanément sur la table. Si d'autres galions sont présents sur la table, ils n'interviennent pas dans les calculs. Les 72 autres valeurs sont organisées en douze groupes modélisant chacun

la situation d'un galion sur la table et constitué de six nombres : l'importance du galion en pièces d'or, la mise maximale sur ce galion [2] et les quatre booléens indiquant si la couleur correspondante est déjà jouée pour ce galion.

Pour cette première version du réseau, le choix s'est reporté sur un réseau à simple couche dans le but de garder une certaine simplicité d'implémentation. La couche de sortie est composée de quatre neurones, chacun représentant l'une des décisions : attaquer, lancer un galion, piocher, ou défausser. Le réseau comporte donc 384 arcs. Pour fonctionner correctement, le RNA doit également respecter les règles du jeu. Pour cela, les neurones de sortie qui correspondent à des actions interdites par le règlement sont inhibés. Par exemple, tant que le talon n'est pas vide, un joueur ne peut défausser. Dans un tel cas, le neurone correspondant à cette action est ignoré dans la prise de décisions.

Les neurones utilisés sont standards (de type McCulloch et Pitts) et leurs fonctions d'activation sont de simples fonctions identités [3]. Une fonction d'activation non-linéaire n'aura qu'un intérêt limité puisque le RNA ne possède pas de couche cachée.

Pour mettre en place l'entraînement du réseau, un ensemble de données d'apprentissage doit être disponible. Or, dans le cas qui nous occupe, une stratégie (c'est-à-dire le comportement du RNA) n'est évaluée qu'en fin de partie sur base des scores des joueurs. La qualité d'une solution donnée par le réseau est difficilemment mesurable car le choix optimal ne pourra finalement être connu qu'à la fin de la partie. Ce fait nous empêche d'utiliser l'algorithme du delta rétro-propagé.

Une manière de contourner cette difficulté consiste à utiliser comme référence les coups effectués par un joueur humain. Cette solution souffre de deux inconvénients. Tout d'abord, le joueur humain est susceptible de commettre des erreurs et faire régresser le RNA. Ensuite, l'algorithme d'apprentissage utilise une grande quantité d'exemples d'entraînement qui obligerait le joueur humain à jouer un nombre de parties très important.

A la suite de ces différentes constatations, nous avons décidé de calibrer

2. Pour chacune des couleurs, la somme des cartes pirates est comptabilisée et la plus grosse valeur est retenue

3. $f(x) = x$

les différentes valeurs des arcs de notre réseau à l'aide d'un algorithme évolutionnaire. Cette méthode nous permet de jouer un nombre important de parties en espérant une convergence des poids du RNA.

Notre implémentation d'un algorithme évolutionnaire sur ce réseau de neurones fonctionne de manière particulière. Premièrement, nous avons volontairement omis la phase de recombinaison car cette opération n'a pas forcément de sens dans le cas du calibrage d'un réseau de neurones.

Notre évaluation des différents candidats est basée sur la proportion de victoires d'un joueur associé dans un ensemble de 1000 parties jouées. La population est articulée autour de n *champions* sélectionnés à l'aide de cette fonction d'évaluation. La génération suivante est composée de mutations de ces champions, ainsi que de joueurs générés aléatoirement. Au terme de l'algorithme, les champions sont mis en compétition afin de retenir le meilleur d'entre eux.

Résultats & Alternatives

Suite à différentes expérimentations, nous avons constaté que le nombre considérable d'arcs dans le réseau ralentissait le temps de traitement global de l'application. En effet, le temps d'initialisation de la couche d'entrée du réseau combiné à la propagation du signal pénalise la rapidité de l'intelligence artificielle. De plus, l'algorithme évolutionnaire n'obtient pas de résultats concluants. Ce problème est dû au nombre important de valeurs à calibrer entraînant ainsi une taille considérable de l'espace de solutions à explorer et pénalisant d'autant la vitesse de convergence d'un tel algorithme. Finalement, les résultats obtenus ne permettent pas d'effectuer une analyse précise car aucune interprétation d'une éventuelle stratégie ne peut être dégagée d'un réseau de neurones de ce type.

La principale limitation de cette conception provient du fait que le réseau ne contient aucune couche interne, ce qui limite sa capacité à refléter la complexité du problème. L'apport de couches internes avec des fonctions d'activations non-linéaires peut améliorer sensiblement le comportement du RNA. Pour être réellement efficaces, ces couches doivent contenir un nombre raisonnable de neurones augmentant ainsi proportionnellement le nombre de poids à calibrer. Dans ce cas, nous atteignons la limite de l'approche méthodologique envisagée, c'est-à-dire la calibration d'un réseau de neurone de taille trop importante via un algorithme évolutionnaire.

Toutefois, il existe un moyen de limiter ce temps d'apprentissage. Comme nous l'avons mentionné ci-dessus, l'apprentissage supervisé basé sur la rétro-propagation via un joueur humain ne fournit pas une solution idéale, mais il peut fournir un bon point de départ à l'algorithme évolutionnaire afin d'améliorer la vitesse de convergence. Cette pré-optimisation couplée à l'utilisation de couches internes permettrait d'améliorer considérablement le comportement de l'intelligence artificielle.

Etant la pauvreté de ces résultats, nous avons envisagé d'améliorer ceux-ci en considérant un réseau multi-couches tout en conservant un nombre raisonnable de poids à calibrer. Pour réaliser cette idée, nous avons pensé à plusieurs petites aggrégations permettant de préciser une valeur pour un ensemble de données. Par exemple, nous pourrions regrouper les arcs provenant des différents neurones symbolisant la main du joueur vers un unique neurone qui synthétise l'information. Ensuite, ces différents neurones *synthétiseurs* sont interconnectés à une couche de sortie. La figure 4.5 présente une illustration de cette méthodologie.

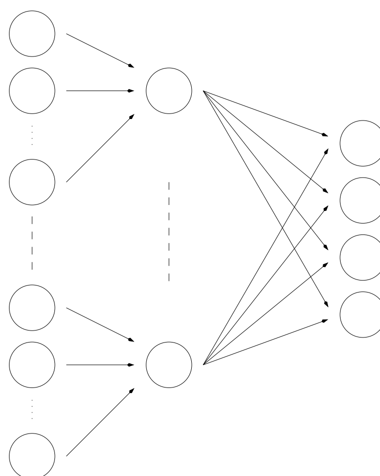

FIG. 4.5 – *Méthodologie à envisager.*

Cette solution résout les problèmes dûs au nombre d'arcs trop important du réseau. De plus, elle permettrait éventuellement d'analyser certains comportements en fonction des poids des arcs liés aux différents agrégats et actions.

Dans la section 2.2.2, nous avons présenté différentes méthodes d'apprentissage non-supervisé (telles que la méthode SOM), qui permettent d'effectuer une classification des données.

La méthode SOM est applicable à notre problème en utilisant une structure sans couche cachée et quatre noeuds de sortie. Une fois son apprentissage terminé, le réseau est capable d'effectuer une classification pour chaque jeu de données reçu en entrée. En d'autres termes, il place chaque situation de jeu dans une des quatres classes définies par l'apprentissage.

Pour transformer ce réseau effectuant de la classification en une intelligence artificielle, il suffit d'associer un type de coups à chaque classe. Par exemple, lorsqu'une certaine situtation de jeu se produit et que celle-ci est incluse dans une certaine classe associée arbitrairement au choix d'attaquer, l'intelligence artificielle prend l'option d'assiéger un galion adverse.

Pour connaître la meilleure association entre les classes et le type de coups à jouer, il suffit de considérer toutes les combinaisons possibles. A chacune de ces combinaisons est associé un joueur et le meilleur d'entre eux est déterminé via une sélection naturelle.

Pour que cette implémentation génère une intelligence artificielle efficace, l'ensemble des données d'apprentissage doit être réparti en quatre classes. Si ce n'est pas le cas, c'est-à-dire si l'ensemble n'a pas de structure particulière ou si les classes formées ne sont pas au nombre de quatre, le comportement du joueur résultant risque de ne pas être très compétitif.

4.2.3 Discussion

Suite à plusieurs expérimentations, nous avons pu constater que l'utilisation d'algorithmes évolutionnaires nous a procuré de meilleurs résultats dans le cas de la calibration d'un arbre de décision probabiliste que dans celui de l'entrainement d'un réseau de neurones. Cependant, l'approche que nous avons eue est peut-être trop simpliste pour représenter un comportement dans un environnement aussi complexe. En effet, la manière dont nous avons travaillé ne nous permet pas de tenir compte d'éventuelles influences entre les paramètres. L'emploi d'un réseau de neurones plus élaboré (multi-couches) permettrait éventuellement d'obtenir de tels résultats. De plus, celui-ci aurait également la possibilité de prendre des décisions plus importantes que le simple choix de l'action à effectuer (choix de la carte à jouer).

Un autre inconvénient de notre méthodologie s'avère être la nécessité de définir certaines stratégies par nos propres moyens. Nous n'avons donc pas la possibilité de préciser celles-ci de manière exhaustive. L'analyse d'une expérimentation avec un réseau de neurones multi-couches permettrait de dégager d'éventuelles stratégies dont nous n'avons pas conscience. Cependant, cette amélioration engendrerait un nombre encore plus important de poids à calibrer, ce qui pénaliserait davantage l'apprentissage. Une autre approche envisagée consiste à agréger les données en entrée afin d'alléger le réseau. Ces deux manières d'aborder le problème n'ont pas été expérimentées mais nous semblent intéressantes pour des études ultérieures.

4.3 Implémentation

Notre implémentation se base sur le langage de programmation *Java*. L'approche *Orienté Objet* se prête bien à la nature du jeu et à sa conception. De plus, celui-ci nous permet de réaliser une interface graphique relativement rapidement.

En premier lieu, nous avons concentré nos efforts sur une distribution aléatoire des cartes du talon. Nous avons associé un identifiant à chaque carte du jeu (cfr Annexe A). Ensuite, il nous a fallu trouver une méthode pour choisir une carte aléatoirement dans cet ensemble et la retirer du talon. Nous avons donc créé une liste de cartes dont nous connaissions la taille n. Par après, nous choisissions un nombre aléatoire dans l'intervalle $[1,n]$ afin de déterminer la carte à donner au joueur. Finalement, nous retirions cette carte de la liste représentant le talon et diminuions n d'une unité.

Ensuite, nous avons réalisé un diagramme de classes du jeu à concevoir permettant d'obtenir une structure de celui-ci dans son intégralité (cfr Figure 4.6). Nous avons conçu une classe pour chaque élément important du jeu : talon, cartes, joueurs et table. Pour les éléments tels que les joueurs ou les cartes, nous avons défini des *interfaces Java* car elle nous permettait d'implémenter deux types de joueurs ou de cartes ayant certaines caractéristiques communes mais dont d'autres diffèrent. Ainsi, il était aisé de sélectionner un joueur de type *algorithme évolutionnaire*, *réseaux de neurones* ou *humain* avant de jouer une partie.

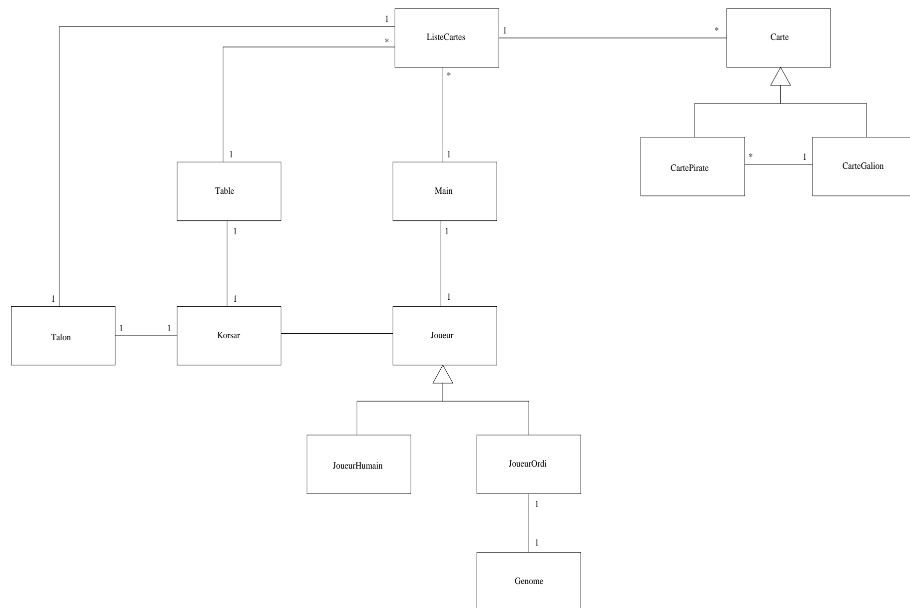

FIG. 4.6 – *Diagramme de classes de Korsar.*

Par la suite, nous avons complété ces différentes classes par des attributs et des méthodes. La première opération fut de donner une carte à un joueur, c'est-à-dire attribuer un numéro de joueur à la carte et ajouter la carte à la main d'un joueur. Afin de pouvoir classer celle-ci, nous avons attribué à chaque main 5 listes de cartes représentant chacune des quatre couleurs de cartes *Pirate*, ainsi que les cartes *Galion*.

Une fois cette opération terminée, nous devions permettre à un joueur de poser une carte *Galion* sur la table. Comme nous l'avons signalé dans le paragraphe précédent, le numéro du joueur est attribué aux cartes ajoutées à sa main afin de pouvoir identifier celles-ci sur la table. Ainsi, il suffit de retirer la carte de la main du joueur pour l'ajouter à la liste des cartes se trouvant sur la table.

Ensuite, il fallait permettre au joueur d'attaquer une carte *Galion* se trouvant sur la table. Pour ce faire, nous devions analyser les différentes cartes éventuellement déjà placées sur cette carte. En effet, comme nous l'avons expliqué lors de la présentation des règles, il faut veiller à ce que chaque couleur ne soit présente qu'une seule fois sur chaque galion. Pour compta-

biliser ces cartes, nous avons associé à celui-ci une liste par couleur de pirates.

Une fois toutes ces possibilités mises en place, nous avons veillé à ce que toutes les règles du jeu soient bien respectées afin de rendre le jeu totalement jouable.

Nous avons ensuite entrepris d'implémenter une interface graphique pour le jeu. Afin de simplifier la mise en place des joueurs autour de la table et d'obtenir une interface similaire à un jeu de cartes standard (belote, whist, bridge, ...), nous avons fixé le nombre de joueurs à quatre. Une fois le talon placé au centre du jeu, nous devions également penser à la disposion des galions et pirates des joueurs sur la table. Afin de ne pas surcharger l'interface, nous avons préféré inscrire sur la table les différentes valeurs des cartes plutôt qu'une image comme nous l'avons fait pour la main des joueurs. Pour chaque galion joué, un joueur disposait d'une structure du type de la figure 4.7 devant sa main de cartes. En se référant à cette figure, nous pouvons analyser plusieurs valeurs. Premièrement, la valeur au centre est la valeur de la carte galion posée par le joueur. Ensuite, les autres valeurs sont associées à une couleur de pirates. Chaque couleur est *dirigée* vers le joueur ayant attaqué le galion avec celle-ci. Par exemple, en se référant à la figure 4.7, si le galion a été joué par la personne à gauche de la table, les cartes pirates jaunes (d'une force totale de 6) appartiennent au joueur situé en face d'elle, c'est-à-dire à droite de la table.

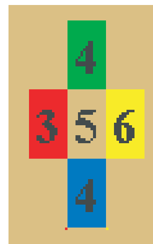

FIG. 4.7 – *Carte Galion et cartes pirates.*

En associant tous ces éléments, nous obtenons une interface telle que présentée à la figure 4.8. Les cartes du joueur en cours sont visibles, tandis que les cartes des autres joueurs sont placées face cachées afin de connaître le nombre de cartes restantes dans leurs mains.

FIG. 4.8 – *Table de jeu de Korsar.*

Dans le but de permettre à un joueur de choisir le galion à attaquer, nous avons créé une petite table de jeu reprenant uniquement la valeur des galions, ainsi que leurs pirates associés (cfr Figure 4.9). Nous donnons la possibilité au joueur de cliquer sur un bouton représentant le galion qu'il désire assiéger.

FIG. 4.9 – *Choix du galion à attaquer.*

Etant donné que plusieurs actions peuvent être effectuées pour une carte, nous avons permis au joueur d'effectuer ce choix via une fenêtre obtenue en cliquant sur la carte en question (cfr Figure 4.10). Les opérations interdites sont représentées sous forme la forme d'un bouton désactivé.

FIG. 4.10 – *Choix d'action à effectuer.*

Finalement, nous avons intégré la possibilité de jouer à plusieurs joueurs humains. A tout moment, une partie peut être relancée en réinitialisant les différents joueurs en fonction des désirs de l'utilisateur (cfr Figure 4.11).

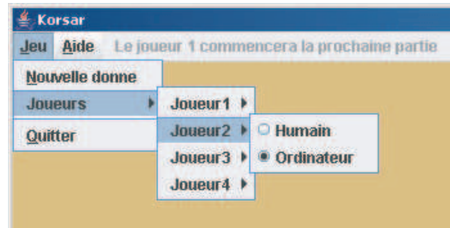

FIG. 4.11 – *Choix du type de joueurs.*

Une aide est également disponible sous la forme d'un règlement de jeu et est accessible via un menu placé en haut de l'interface (cfr Figure 4.12).

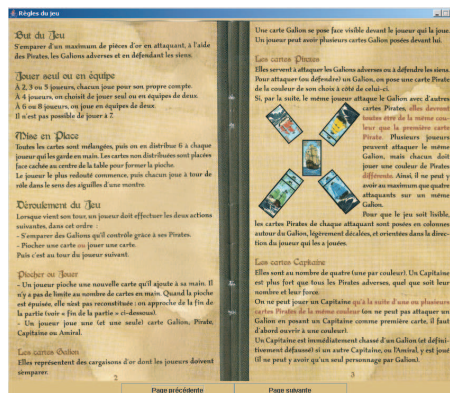

FIG. 4.12 – *Règles du jeu.*

Conclusion

Les différentes techniques étudiées ont permis de réaliser avec succès divers moteurs d'intelligence artificelle dans le cadre d'un jeu de gestion multijoueurs. Ceux-ci offrent une alternative raisonnable à un être humain car leurs niveaux se rapprochent d'un joueur moyen.

Néanmoins, ces implémentations ont mis en lumière certaines limites de notre méthodologie. En effet, s'ils sont de bonne qualité, les joueurs artificiels ne peuvent pas toujours rivaliser avec un joueur humain expérimenté.

L'amélioration du comportement des joueurs passe par une vision différente de la manière d'aborder le problème. Cette approche doit offrir la possibilité de modéliser une gamme de comportements plus riche tout en veillant à limiter le nombre de paramètres à calibrer lors d'un apprentissage par la machine. Une combinaison plus élaborée des techniques présentées ou l'emploi d'une toute autre méthodologie comme la programmation par agents ou les réseaux bayésiens pourrait constituer un domaine d'expérimentation intéressant.

Bibliographie

[1] Ph. TOINT & C. ROEGIERS, *Algorithmes évolutionnaires pour la programmation non-linéaire globale*, FUNDP Namur, 2004.

[2] T. MITCHELL, *Machine Learning*, The McGraw-Hill Companies, 1997.

[3] J.-M. ALLIOT & T. SCHIEX, Intelligence artificielle & Informatique théorique, Cepadues, 1 mai 2002.

[4] A. HARDY, *Syllabus de Classification*, FUNDP Namur, 2000.

[5] G. DREYFUS, *Réseaux de neurones : Méthodologie et applications*, Eyrolles, 2004.

[6] J.-B. MOURET, *Concepts fondamentaux des algorithmes évolutionnistes*, 15 novembre 2005.

[7] J.-B. MOURET, *Algorithmes évolutionnistes, deuxième partie : évolution artificielle de créatures*, 15 novembre 2005.

[8] I. RIVALS, *Les réseaux de neurones formels pour le pilotage de robots mobiles*, ESPCI, 1996.

[9] O. BOISSIER, *Systèmes multi-agents*, Mines Saint-Etienne, 2001.

[10] A. FLOREA, *Agents et Systèmes multi-agents*, Politechnica University of Bucharest, 2002.

[11] M. LABOUS, *Le CNES travaille sur l'intelligence des robots*, CNES Magazine Toulouse.

[12] S. DONCIEUX & J.-A. MEYER, *Evolution of neurocontrollers for complex systems : alternatives to the incremental approach*, Animatlab - LIP6, France, 2004.

[13] H. BERGSON, *L'évolution créatrice*, Chicoutimi, Québec, 2003.

[14] B. GIRARD & G. ROBERT & A. GUILLOT, *Jeu Vidéo et Intelligence Artificielle Située*, Animatlab - LIP6, France, 2002.

[15] R. COULON, *Des réseaux de neurones artificiels apprennent la natation.*

[16] J. QUINQUETON, *Fondements des Systèmes Multi-Agents*, Montpellier, France.

[17] J. QUINQUETON, *Organisation des Systèmes Multi-Agents*, Montpellier, France.

[18] F. MONDADA & D. FLOREANO, *Conception évolutionniste de réseaux de neurones pour le contrôle de robots mobiles*, Ecole polytechnique fédérale de Lausanne.

[19] O. ROGÉ, *Création de joueurs virtuels pour un simulateur pédagogique*, Université René Descartes, 2003.

[20] J. KODJABACHIAN, *Développement et évolution de réseaux de neurones artificiels*, Université Pierre et Marie Curie, 16 février 1998.

[21] L. FISCHER, *L'intelligence Artificielle dans les jeux*, Joystick n°141, Octobre 2002.

[22] D. GOLDBERG, *Genetic Algorithms*, Addison Wesley, 1989.

[23] M. BUCKLAND, *AI Techniques for game programming*, Premier Press Cincinnati, Ohio, 2002.

[24] C. DUNIS & M. WILIAMS, *Modelling and Trading the EUR/USD Exchange Rate: Do Neural Network Models Perform Better?*, Liverpool Business School, 2002.

[25] R. SHARDA, D. DELEN , *Predicting box-office success of motion pictures with neural networks*, Elsevier, 2005.

[26] D. MACKAY , *Information Theory, Inference,and Learning Algorithms*, Cambridge University Press , 2003.

[27] B. KRÖSE & P. VAN DER SMAGT, *An introduction to Neural Networks*, University of Amsterdam, 1996.

[28] B. KRÖSE & P. VAN DER SMAGT, *Programming lab for the Neural networks course*, University of Amsterdam, 1994.

[29] W. GERSTNER , *Supervised Learning for Neural Networks: A Tutorial with JAVA exercices*, EPFL.

[30] P. TINO , *Neural Network Applications*, School of Computer Science University of Birmingham.

[31] http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

[32] http://www.scico.u-bordeaux2.fr/~corsini/Cours/mainRNA/mainRNA.html

[33] http://www.dacs.dtic.mil/techs/neural/neural_ToC.html

[34] http://www.neurocomputing.org/History/history.html

[35] http://diwww.epfl.ch/~gerstner/SPNM/SPNM.html

[36] http://cortex.snowseed.com/neural_networks.htm

[37] http://www.neural-networks-at-your-fingertips.com/

[38] http://www.cs.cmu.edu/Groups/AI/html/faqs/ai/neural/faq.html

[39] http://www.cs.stir.ac.uk/~ lss/NNIntro/InvSlides.html
[40] http://www.wikipedia.org/
[41] Le Petit Larousse illustré 1999 (Version CD-Rom)
[42] http://news.bbc.co.uk/nolpda/ukfs_news/hi/newsid_4534000/4534451.stm
[43] http://diwww.epfl.ch/~ gerstner/BUCH.html
[44] http://turing.cs.pub.ro/auf2/
[45] http://www.dataligence.com/newsartificialintelligence.phtml?cat=25
[46] http://gilco.inpg.fr/~ rapine/Graphe/Cheminement/recherche.html
[47] http://www.emn.fr/x-info/pdavid/Enseignement/IA/poly-ia/jeux/introduction.html
[48] http://njussien.e-constraints.net/sudoku/
[49] http://animatlab.lip6.fr/
[50] http://www.supinfo-projects.com/2005/recherche_heuristic_echec/
[51] http://pagesperso.laposte.net/autismeprehistoire/reseaux_de_neurones.htm
[52] http://www.lesrobots.com/robots/neurones_artificiels.htm
[53] http://www.contrib.andrew.cmu.edu/~ rjg/millibots/millibot_project.html
[54] http://www.automatesintelligents.com/echanges/2002/jan/billet3.html
[55] http://www.emn.fr/x-info/pdavid/Enseignement/IA/poly-ia/jeux/arbre-jeu-tic-tac-toe.html
[56] http://www.futura-sciences.com
[57] http://afia.lri.fr/node.php?lang=fr&node=1008
[58] http://ai-news.elzemozgurce.net/
[59] http://www.vieartificielle.com
[60] http://www-neos.mcs.anl.gov/
[61] http://www.auml.org/

Annexes

Annexe A

Identifiants des cartes

A.1 Cartes Galion

Les 5 cartes *Galion* de valeur	2 portent les numéros allant de	0 à 4.
6	3	5 à 10.
5	4	11 à 15.
5	5	16 à 20.
2	6	21 à 22.
1	7 porte le numéro	23.
1	8 porte le numéro	24.

A.2 Cartes Pirates

Nous avons classé les couleurs de la manière suivante : Rouge, Bleu, Vert, Jaune

A.2.1 Couleur Rouge

Les 2 cartes *Pirates* ROUGE de valeur	1 portent les numéros allant de	25 à 26.
4	2	27 à 30.
4	3	31 à 34.
2	4	35 à 36.

A.2.2 Couleur Bleu

Les 2 cartes *Pirates* BLEU	de valeur 1 portent les numéros allant	de 37 à 38.
4	2	39 à 42.
4	3	43 à 46.
2	4	47 à 48.

A.2.3 Couleur Vert

Les 2 cartes *Pirates* VERT	de valeur 1 portent les numéros allant	de 49 à 50.
4	2	51 à 54.
4	3	55 à 58.
2	4	59 à 60.

A.2.4 Couleur Jaune

Les 2 cartes *Pirates* JAUNE	de valeur 1 portent les numéros allant	de 61 à 62.
4	2	63 à 66.
4	3	67 à 70.
2	4	71 à 72.

Annexe B

Documents disponibles

L'implémentation réalisée dans le cadre de ce travail est présente sur un cédérom joint à ce document. Elle comporte les différentes classes Java, ainsi que la JavaDoc de l'application.

Nous y avons également inclus les logiciels d'installation du *Java Development Kit 5.6* et *Java Runtime Environment 5.6* afin de fournir l'environnement nécessaire à notre application.

Enfin, une version électronique de ce document est présente sous format *postscript* et *pdf.*