République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université 20 aout 1955 Skikda

Faculté de sciences – département d'informatique

Mémoire de fin d'étude en vue de l'obtention du diplôme

Master académique en Informatique

Option :Réseaux Et Systèmes Distribues

Thème :

L'application d algorithme génétique pour l'optimisation d'énergie dans les réseaux de capteurs sans fil

Présenté par :

- Kiouaz Mouna
- Touati Rayane

encadre par:

- Dr Benoudina Lazhar

Année : 2022/2023

Remerciement

Nous remercions tout d'abord Allah , le tout puissant de nous avoir illuminé et ouvert les port de savoir et nous avoir donné la volonté et le courage d'élaborer ce travail .

Nous tenons à exprimer vivement notre profonde gratitude à notre promoteur Monsieur : Benoudina Lazhar pour sa confiance, ses encouragements, ses merveilles corrections et pour les conseils qu'il a apporté pour l'achèvement de ce projet.

Nous tenons également à remercier l'ensemble de membres de jury qui nous ont fait l'honneur de juger notre travail.

Nous tenons aussi à exprimer nos remerciements à tous ceux qui nous ont aidés de près de loin l'élaboration de notre mémoire de fin d'étude.

Liste des Figures

Introduction générale

Chapitre 1 : Introduction à la théorie des jeux

Chapitre 2 : Algorithmes évolutionnaire : Principes et méthodes

Chapitre 3 : Introduction aux Réseaux de neurones

Chapitre 4 : Analyse et Conception

Chapitre 5 : Implémentation

Liste des tableaux

Table des matières

Introduction générale

Chapitre 1 : Introduction à la théorie des jeux

Chapitre 2 : Algorithmes évolutionnaire : Principes et méthodes

Chapitre 3 : Introduction aux Réseaux de neurones

Chapitre 4 : Analyse et Conception

Liste des algorithmes

RÉSUMÉ :

Un grand nombre de travaux ont été effectués sur l'évolution des réseaux de neurones par les algorithmes évolutionnaires. Ces derniers, peuvent être employés pour la construction et l'apprentissage de réseaux neuronaux tant du point de vue structure que du point de vue des pondérations. La réalisation de notre projet est fondée sur l'hybridation des algorithmes évolutionnaires avec les réseaux de neurones pour faire évoluer une population de stratégies de jeu. Un algorithme évolutionnaire s'est enseigné comment jouer le jeu de dames sans utiliser des caractéristiques qui exigeraient l'expertise humaine. En utilisant seulement les positions de pièces sur la table de jeu et le différentiel de pièce, le programme évolutionnaire a optimisé des réseaux de neurones artificiels pour évaluer des positions alternatives dans le jeu. Pour la durée de plusieurs centaines de générations, le programme s'est enseigné à jouer à un niveau qui est compétitif avec des experts humains.

MOTS-CLÉS : Théorie des jeux, Hybridation, Algorithmes évolutionnaires, Réseaux de neurones, Algorithmes génétiques, Stratégies d'évolution, Algorithmes de recherche, Minimax, Alpha bêta, Fail-Soft Alpha-Bêta, Apprentissage, rétro-propagation du gradient.

DISCIPLINE ADMINISTRATIVE: Informatique.

ABSTRACT:

A large number of works were made on the evolution of neuronal networks by evolutionary algorithms. The latter, can be used for the construction and the apprenticeship of neuronal networks so much point of view structures that from the point of view of the level-weightings. The realization of our project is based on the hybridization of the evolutionary algorithms with the neuronal networks to develop a population of strategies of game. An evolutionary algorithm taught how to play the checkers without using characteristics which would require the human expertise, only by using the positions of parts on the gaming table and the difference of part. The evolutionary program optimized artificial neuronal networks to estimate alternative positions in the game. For a period of several hundreds of generations, the program taught to play a level which is competitive with human experts.

KEYWORDS: Theory of the games, Hybridization, evolutionary Algorithms, neuronal networks, Algorithms genetics, Strategies of evolution, Algorithms of search, Minimax, Alpha bêta, Fail-Soft Alpha-Bêta, Apprenticeship, retro-propagation of the gradient.

ADMINISTRATIVE DISCIPLINE: Computer Science

INTRODUCTION GÉNÉRALE

1. Contexte général :

Qui parmi nous n'a jamais participé aux jeux d'échecs, jeux de dames, jeux de cartes ou tout simplement ne s'est jamais trouvé dans une situation de concurrence. Mais personne n'a jamais imaginé que ce moment de distraction ou cette situation de rivalité pouvait être présentée sous forme d'un modèle mathématique. Cela se fait par les concepts de la *théorie des jeux*.

La *théorie des jeux* est une discipline mathématique qui étudie les situations où l'état final des individus (joueurs) dépond non seulement des décisions qu'il prend mais également des décisions prises par l'adversaire. Ainsi, le choix *optimal* pour un individu dépend généralement de ce que fait son adversaire. En effet, la théorie des jeux a développé une vocation pour les sciences sociales et économiques, avec des applications disparates. Elle apparait aujourd'hui comme un paradigme très général de concepts et de techniques, dont le potentiel reste encore à exploiter en informatique. En fait, les *jeux* peuvent être chose sérieuse. Ils sont un moyen d'évaluer l'intelligence des ordinateurs par rapport à la nôtre.

2. Problématique :

Les problèmes d'*optimisations* occupent actuellement une place de choix dans la communauté scientifique. Non pas qu'ils aient été un jour considérés comme secondaires mais l'évolution des techniques informatiques a permis de dynamiser les recherches dans ce domaine. [1]

Le comportement des stratégies alternatives dans les jeux est définit par le tracé du stimulus-réponse de chaque individu. La limitation de ces comportements aux fonctions linéaires des conditions environnementales rend les résultats douteux. En effet, la prise des décisions efficaces dans n'importe quel environnement complexe exige presque toujours un tracé non linéaire du stimulus-réponse.

Dans le cadre de notre mémoire, l'obstacle vient dans le choix de la représentation des stratégies de jeu appropriées et de l'algorithme du recherche utiliser ainsi que sa fonction d'évaluation afin de remédier aux problèmes d'*explosion combinatoire* qui reste un défait pour les gents qui s'intéressent de la théorie des jeux combinatoire.

3. Approche adoptée :

L'hybridation de plusieurs méthodes afin d'exploiter les avantages de chacune, semble constituer une approche intéressante pour surmonter ces difficultés. La combinaison la plus adéquate dans notre cas est l'hybridation des réseaux de neurones et les algorithmes évolutionnaires (EA). La principale raison est que ces deux méthodes présentent des avantages complémentaires, elles fournissent des outils pour la résolution de problème de sélection de l'action dans les jeux.

Les EA ont aussi montré leur capacité à éviter la convergence des solutions vers des optimums locaux, aussi bien lorsqu'ils sont combinés avec des méthodes de *recherche locale* comme la rétro-propagation du gradient que lorsqu'ils sont seuls. [2]

La plupart des problèmes peuvent être résolus par une méthode de type recherche locale. C'est le cas, de l'apprentissage d'un réseau de neurones par rétro-propagation du gradient. Partant d'une configuration de poids initiale, la méthode va chercher la meilleure solution dans le voisinage de cette configuration. [3]

Cette solution est optimale localement mais peut ne pas correspondre à un *optimal global* car il est possible qu'il existe une meilleure solution qui n'est pas dans le voisinage de la configuration initiale (figure I.1). [4]

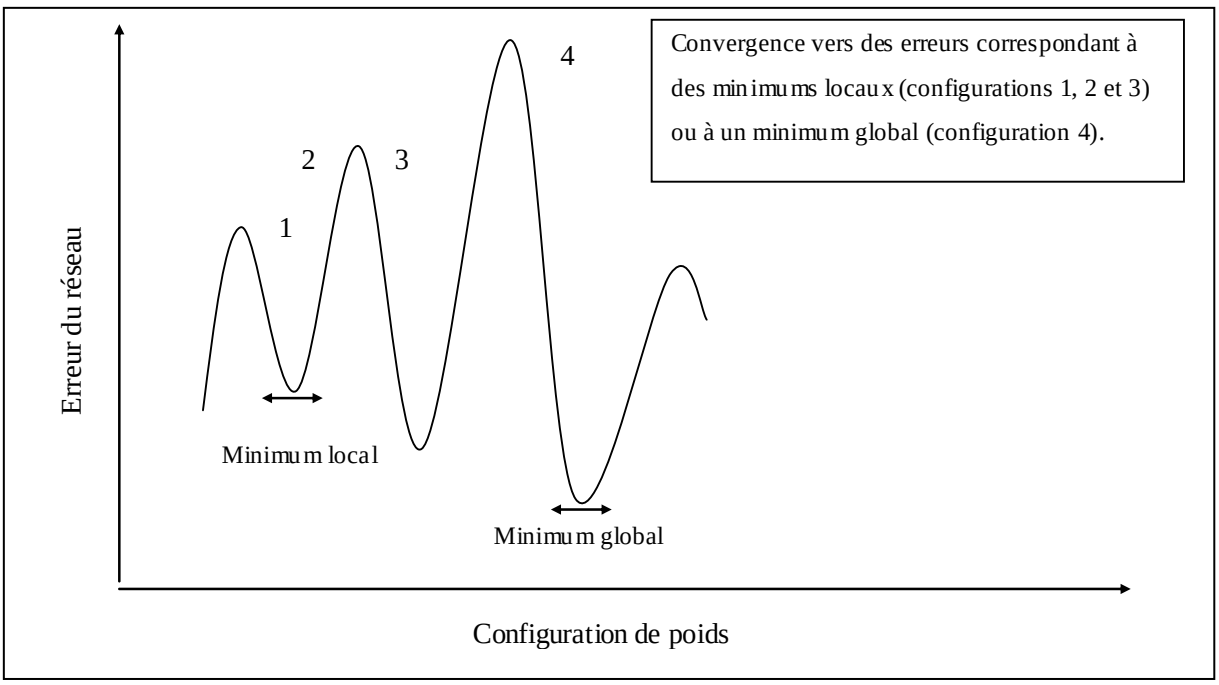


Figure I.1 : Exemple de convergence locale ou globale d'un réseau de neurones.

L'arrivé des stratégies d'évolution permet d'utiliser l'ordinateur comme un outil pour découvrir des stratégies de jeu où l'heuristique peut être indisponible. De même, l'utilité de mettre en application les réseaux de neurones comme fonction d'évaluation non linéaire pour produire des stratégies dans les jeux complexes devient évidente. Alors, l'hybridation des stratégies d'évolution avec les réseaux de neurones semble bien adaptée à découvrir des stratégies optimales dans les jeux où la théorie des jeux classique est incapable de fournir des décisions efficaces.

Le présent travail est l'un des thèmes de recherche qui a été initié par deux chercheures Américains *David B.Fogel* et *Rumar Chellapilla* [5], et qui a pour objectif d'implémenter l'hybridation du calcul neuronal avec le calcul évolutionnaire pour découvrir de nouvelles stratégies créées d'une façon aléatoire.

L'intérêt de l'hybridation se base donc sur l'observation qu'une recherche globale effectuée par des algorithmes évolutionnaires. Les méthodes de recherche locales sont sujettes à des variations de performances dues à la configuration initiale du réseau de neurones menant parfois à une convergence vers des optima locaux. Les méthodes évolutionnaires à l'opposé, assurent une recherche dans le domaine complet. Au fur et mesure des générations cet espace de recherche est affiné vers des sous espaces potentiellement performants. Cependant, il est courant pour les algorithmes évolutionnaires de trouver une solution proche de la meilleure sans jamais l'atteindre.

4. Plan du mémoire :

Le travail présenté dans ce mémoire a été organisé en cinq chapitres :

Le *premier chapitre* est consacré à l'étude de la théorie des jeux en indiquant principalement les différents types de jeu ainsi que les méthodes de recherches dans les arbres de jeu (Minimax, Alpha-Beta, Fail soft Alpha-beta, …).

Dans le *second chapitre*, nous abordons les concepts de base des algorithmes évolutionnaires en se concentrant sur les études des stratégies d'évolution qui seront utilisées dans l'évolution des réseaux de neurones.

Dans le *troisième chapitre*, on va étudier une des classes de méthodes d'apprentissage automatique qui sont les réseaux de neurones. On va voir qu'est ce qu'un neurone, qu'est ce

qu'un réseau de neurones, les types dernières, les différentes méthodes d'apprentissage utilisé dans ces méthodes.

Le *quatrième chapitre* présente notre travail "L'hybridation du calcul neuronal et du calcul évolutionnaire" pour faire améliorer les stratégies des jeux. On va montrer d'une manière détaillée les étapes de conception de notre projet. Il s'agit d'une méthode pour améliorer les stratégies dans les jeux de dames dont nous expliqueront ces règles ainsi que la description de la méthode évolutionnaire utilisée pour l'amélioration. Nous traitons aussi un cas particulier de réseau de neurones appelé le Perceptron Multicouches choisie comme modèle de base dans notre travail.

Le *dernier chapitre* est consacré pour la représentation de l'étape de l'implémentation de notre logiciel.

Pour terminer, une *conclusion générale* résumant le travail, et une exposition de quelques perceptives, notamment celles à l'utilisation de l'hybridation du calcul neuronal avec le calcul évolutionnaire dans plusieurs domaines autre que les jeux, ainsi que l'optimisation du temps d'exécution des algorithmes.

CHAPITRE I

INTRODUCTION Á LA THÉORIE DES JEUX

Ce chapitre présente les principes fondamentaux de la théorie des jeux qui permettent la modélisation mathématique des jeux, ce qui facilite la tâche de la programmation, tout en explorant les algorithmes de recherche les plus répandus pour représenter et optimiser les stratégies des jeux.

1. Introduction :

Dans de nombreuses situations de la vie quotidienne, la performance d'un acteur, qu'il soit un individu, une entreprise ou un pays, ne dépend pas uniquement de son action, mais aussi de celle prises par les autres. Cette interdépendance stratégique est le domaine de prédilection de *la théorie des jeux*.

La théorie des jeux est une branche des mathématiques, de la recherche opérationnelle et de l'économie, qui a pour objet la modélisation mathématique des situations conflictuelles. Elle propose des concepts opératoires et des outils formels pour l'analyse du conflit ainsi que sa représentation.

En fait, elle consiste à analyser l'interaction dans un groupe d'agents [1] rationnels [2] qui a un comportement stratégique, par exemple elles nous permettent de mieux comprendre le déroulement des guerres. [6]

Cette théorie définit une étude des comportements rationnels des individus en situation de conflit, à travers des modèles appelés *jeux*. Elle a été appliquée pour la première fois en science économique où elle a remporté un franc succès. Par la suite, on s'est aperçu que les jeux sont présents dans des domaines aussi inattendus que la biologie, la sociologie, et l'informatique. [7]

2. Historique : [6]

Certaines idées de base de la *théorie des jeux* aient été présentées avant même sa naissance à travers les écrits de *Cournot*, *Zermelo* et d'*Emile Borel*, le résultat pionnier de cette théorie revient à *John Von Newman* en 1928. Ce dernier a démontré le théorème de min-max qui a joué et joue jusqu'à présent un rôle très important dans la théorie des jeux.

C'était en 1944, dans un ouvrage très réputé intitulé « *Game Theory and Economic Behavior* » que le mathématicien *John Von Newman* et l'économiste *Oskar Morgenstern* ont donné naissance à la théorie des jeux.

---

[1] Groupe d'agents. Toute personne qui participe au conflit et capable de prendre une décision, appelé aussi joueur.

[2] Rationalité. La rationalité individuelle d'un joueur est une règle de maximisation du gain individuel.

En répondant aux insuffisances d'équilibres, utilisés dans la microéconomique traditionnelle, *John Nash* a mis en évidence la notion de l'***équilibre de Nash*** qui prend comme référence le principe de la rationalité individuelle.

Cette notion peut conduire chaque individu à une situation de non regret mais elle ne peut pas lui garantir un gain optimal.

La théorie des jeux a été, enfin, consacrée par l'obtention du prix de ***Nobel d'économie***, en 1944, attribué aux trois chercheurs *J. Nash, C. Harsanyi* et *R. Selten*. Ces derniers ont contribué à faire avancer la science économique.

Assez rapidement, cette théorie a été considérée comme une solution éventuelle aux problèmes de formalisation que connaissaient les sciences économiques, les sciences sociales, les situations politiques, militaires et autres.

3. La théorie des jeux classique :

La théorie des jeux prend comme hypothèse principale la *rationalité* forte des individus "*Chaque individu cherche à maximiser ses gains personnels en prenant en considération le comportement de ses adversaires*". La théorie classique constitue une approche mathématique des différentes stratégies de chacun des individus et cherche à trouvé une solution optimale pour résoudre les conflits. Dans ce cadre, les théoriciens des jeux ont introduit la notion d'équilibre. [7]

Tout jeu comporte une liste de joueurs, un ensemble de stratégies possibles pour chacun, et des règles qui donnent les gains des joueurs. Chaque choix stratégique d'un joueur a un impact sur les gains d'un autre joueur, et on parle donc aussi de « *théorie de la décision en* interaction ». [8]

4. La théorie des jeux évolutionniste :

Une branche particulière de la théorie des jeux, développée par des biologistes de l'évolution à partir des années 70, s'est détachée de la théorie initiale classique, on parle ici de la *théorie des jeux évolutionnistes*. Les biologistes ont utilisé les principes de la théorie des jeux pour modéliser certains aspects de l'évolution biologique. [8]

En théorie des jeux évolutionnaire chaque individu cherche à améliorer non pas son gain personnel mais le gain total de la population dont il fait partie, son avantage est d'éviter le

problème majeur de la théorie des jeux classique : *la caractérisation des comportements rationnels* et *la nécessité d'anticiper les actions des autres joueurs*.

Dans les jeux évolutionnistes toute idée de choix stratégique et d'anticipation - et donc de rationalité - est abandonnée, ce n'est plus la rationalité de chaque individu qui le pousse à adapter son comportement aux stratégies de ses adversaires, mais une évolution propre à l'ensemble de la population à laquelle il appartient, et dont il est simplement un acteur parmi d'autres. [7]

5. Définition d'un jeu :

Un jeu est une situation où des individus (joueurs) sont conduits à faire des choix parmi un certain nombre d'actions possibles, et dans un cadre défini à l'avance "*les règles du jeu*", qui permet de déterminer qui peut faire quoi et quand. Les résultats de ces choix constituent une issue du jeu à laquelle est associé un gain pour chacun des participants. Ces résultats ne dépendent pas de la décision d'un seul joueur et ne dépendent pas non plus uniquement du hasard, bien que celui-ci puisse intervenir. [7]

En fait, un jeu forme un petit univers plus facile à maîtriser que certains problèmes réels, mais quand même suffisamment complexe pour que l'on puisse faire intervenir des notions typiquement humaines comme *la réflexion* et *le raisonnement*. [9]

6. Les types des jeux :

Les jeux sont souvent différenciés par leur appartenance, ou leur non appartenance, à une des catégories suivantes :

6.1. Jeux finis :

Un jeu est dit ***fini***, s'il satisfait aux conditions suivantes :

- Il est joué en un nombre fini de coups.
- Chaque joueur a un nombre fini de choix à chaque coup.

***Exemple* :** Jeu de Nim. (Vider plusieurs tas de pions de taille variable - traditionnellement, il y a trois tas de 3, 4 et 5 pions). On ne peut enlever qu'un nombre fini d'allumettes, et on ne peut jouer que nombre d'allumettes coups au maximum.

6.2. Jeux à somme nulle :

Un jeu à ***somme nulle*** est un jeu où les paiements sont réciproques. C'est à dire que les gains d'un joueur sont les pertes d'autres joueurs. Il n'y a pas d'apport de l'extérieur.

Exemple **:** Jeu de Dames. Si un joueur gagne, c'est grâce à la défaite de l'autre.

6.3. Jeux à information parfaite/imparfaite :

Un jeu à ***information parfaite*** est un jeu satisfaisant aux conditions suivantes :

- Les joueurs jouent successivement.
- Chaque joueur est parfaitement renseigné sur les coups précédents des autres joueurs.

Exemple **:** Jeu d'Echecs est à information *parfaite*. Les joueur jouent à tour de rôle et connaissent tous les mouvements depuis le début de la partie.

Un jeu est à ***information imparfaite*** si :

- Un des joueurs ne connaît pas, à un moment du déroulement du jeu, ce qu'a joué un autre joueur. Ceci peut arriver dans le cas où on cache l'information aux joueurs ou parce que les joueurs jouent simultanément.

Exemple **:** Jeu du dilemme du prisonnier est à information *imparfaite* car les deux joueurs jouent simultanément.

6.4. Jeux à information complète/incomplète :

On dit qu'un jeu est à ***information complète*** si chaque joueur connaît lors de la prise de décision :

- Ses possibilités d'action.
- Les possibilités d'action des autres joueurs.
- Les gains résultants de ces actions.
- Les motivations des autres joueurs.

Exemple **:** Le jeu du dilemme du prisonnier est à information *complète* car chacun des prisonniers connaît parfaitement la règle du jeu définie par le policier ainsi que l'utilité de l'autre joueur.

Le jeu est dit à ***information incomplète*** si :

- Au moins un des joueurs ne connaît pas entièrement la structure du jeu.

Exemple **:** jeu de cartes.

6.5. Jeux coopératifs/non coopératifs :

Les jeux ***coopératifs*** sont les jeux dans lesquels on cherche la meilleure situation pour les joueurs sur des critères tels que *la justice*. On considère qu'ensuite les joueurs vont jouer ce qui aura été choisi, il s'agit d'une approche normative.

Exemple **:** jeu de football.

On appelle jeu ***non coopératif***, tout jeu où les joueurs ne peuvent pas se regrouper en coalitions, ils peuvent être d'accord sur telle ou telle issue, à condition qu'ils ne contractent pas d'accord contraignant. Aucun joueur ne cherchera à manipuler les autres, il ne cherche qu'à maximiser son propre gain.

7. Les types des joueurs :

Afin de mieux comprendre les principes de la théorie des jeux, il faut toujours avoir à l'esprit les deux caractéristiques fondamentales de joueurs théoriques déroulant les algorithmes. Ils sont supposés :

7.1. Joueur intelligent :

Dire qu'un joueur est ***intelligent***, revient à dire qu'il jouera toujours le meilleur coup. Il ne commettra donc jamais d'erreur et jouera toujours le coup l'avantageant le plus. [10]

7.2. Joueur prudent :

Les joueurs ***prudents*** ne prennent pas de risques dans le but de gagner plus. Ils cherchent toujours à minimiser leurs pertes potentielles. [10]

Remarque **:** Pour plus de lisibilité, Il est nécessaire d'adopter dans la suite de ce travail, une notation commune et compréhensible pour décrire les joueurs : On prend donc arbitrairement, le joueur qui fait le premier coup est appelé ***joueur maximisant***, l'autre est appelé ***joueur minimisant***.

8. Les stratégies des jeux :

8.1. Généralités :

Une stratégie est un plan d'actions complet pour chaque joueur spécifiant ce que fera ce dernier à chaque étape du jeu et face à chaque situation pouvant survenir au cours du jeu.
La stratégie décrit totalement le comportement d'un joueur.

Elle peut-être représentée par une série de "*Si…alors…sinon*", prenant en considération toutes les situations possibles. Une partie de jeu se modélise donc en deux coups, c'est à dire les choix de stratégies des deux joueurs. Une fois choisie, une stratégie ne peut être changée, elle détermine le déroulement de toute la partie pour le joueur concerné.

On peut alors se demander quel est le nombre maximal de stratégies dont peut disposer un jeu. Certaines peuvent être comptées plusieurs fois mais ce dénombrement porte sur les façons dont le jeu peut se dérouler et non pas sur les stratégies.

***Exemple* :** Jeu d'échecs : on ne peut pas définir toutes les stratégies possibles.

Remarque :

- Il n'est possible de connaître l'ensemble des stratégies que pour très peu de jeux.
- Dans la plupart des cas, il est impossible de représenter une stratégie dans son intégralité sous une forme rapidement compréhensible (une suite de tests de positions est assez peu digeste à représenter).

8.2. Les types des stratégies : [7]

Il existe deux types de stratégies :

8.2.1. Stratégies pures :

Une stratégie est dite pure si elle ne contient aucune notion d'aléatoire et n'utilise pas des fonctions de probabilité.

8.2.2. Stratégies mixtes :

Ce sont les stratégies qui consistent à donner une distribution de probabilité sur les différentes actions possible.

***Exemple* :** "*Il sait que je sais qu'il sait que je vais appliquer telle stratégie*". Afin d'éviter ce type de raisonnement au nième degré, on a recours aux stratégies mixtes, dont le principe est d'affecter une probabilité d'être jouée à chaque stratégie, en privilégiant la meilleure stratégie déterminée par le minimax. [10]

8.3. Choix de stratégies :

Lorsqu'une *stratégie optimale* existe, elle est choisie. Lorsqu'elle n'existe pas, ou lorsque plusieurs stratégies équivalentes sont disponibles, on effectue un *choix aléatoire*.

9. Représentation des jeux :

Afin de rechercher le meilleur coup, ou la meilleure stratégie, il est nécessaire de représenter un jeu ou une partie d'une manière exploitable. Les deux méthodes de représentation sont :

9.1. Représentation matricielle :

Appelé aussi *forme normale*, ce mode de représentation se situe au niveau des stratégies, dont il modélise les effets ou les résultats.

Dans ce mode de représentation, les lignes et les colonnes représentent les stratégies ouvertes aux deux joueurs. Par convention, les lignes représentent généralement les stratégies du joueur maximisant. Par extension, les colonnes représentent celles du joueur minimisant.

Les valeurs des différentes cases représentent la valeur d'une partie issue de la confrontation des stratégies de la colonne et de la ligne correspondantes. [10]

***Remarque* :** Les valeurs des cases sont à considérer du point de vue du joueur maximisant.

***Exemple* :**

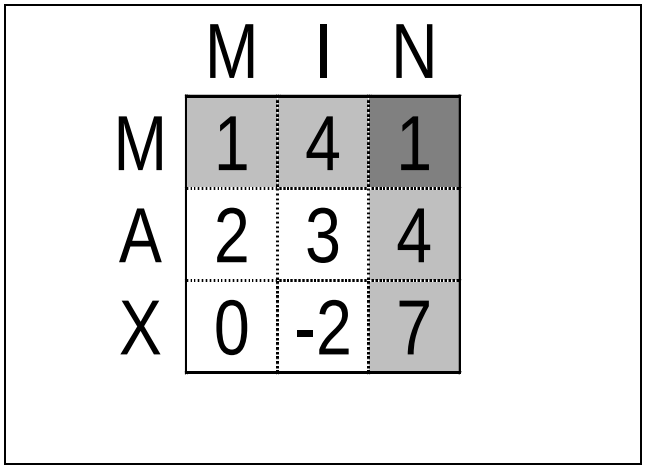


Figure 1.1 : Exemple de représentation matricielle d'un jeu

Dans cet exemple, si le joueur maximisant choisit la stratégie correspondant à la première ligne, et le joueur minimisant celle correspondant à la troisième colonne, alors le résultat de la partie sera 1, correspondant à un gain du joueur maximisant.

***Remarque* :** Les cases correspondent aussi aux feuilles de l'arbre de jeu.

9.2. Représentation arborescente :

Appelé aussi *forme extensive*, Ce mode de représentation se situe au niveau des coups et des positions produites. C'est une représentation explicite de toutes les actions possibles du jeu.

À chaque niveau on a tous les choix possibles pour un joueur, pour un coup donné.

- Les nœuds représentent *les positions* du jeu. Ainsi, la racine est la *position initiale* du jeu, et les feuilles, ou nœuds terminaux correspondent aux *positions de fin de partie*.
- Les arcs représentent *les coups* d'un joueur.
- Les nœuds du premier niveau représentent donc les positions que peut atteindre le premier joueur en un déplacement. Ainsi, chaque chemin partant de la racine vers un nœud terminal représente une partie différente, complète, du jeu.

***Exemple* :**

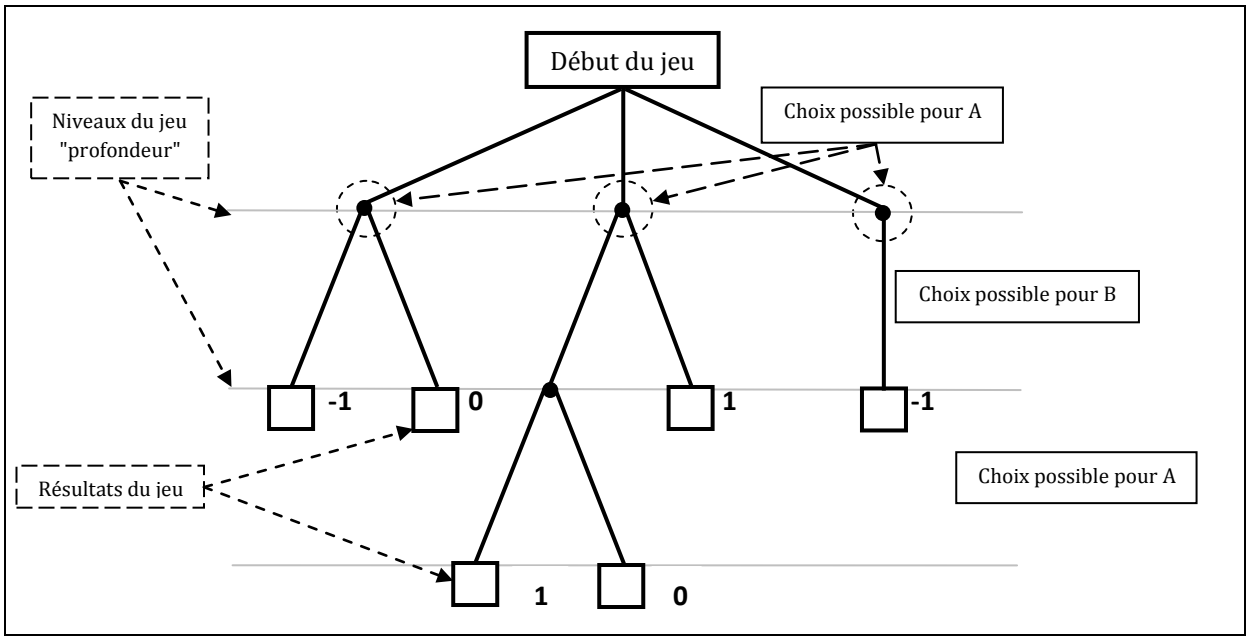


Figure 1.2 : Exemple de représentation arborescente d'un jeu.

***Remarque* :** La représentation d'un arbre de jeu complet se heurte rapidement à une *explosion combinatoire*, tout du moins pour les jeux "intéressants".

***Exemple* :** Le jeu d'échecs, le niveau 2 contient déjà 400 nœuds.

Remarque* :** dans la suite, nous allons adopter la représentation suivante: Les nœuds représentant des positions où le joueur maximisant doit jouer seront représentés par des ***carrés, les autres (joueur minimisant) seront représentés par des ***cercles***.

***Exemple* :**

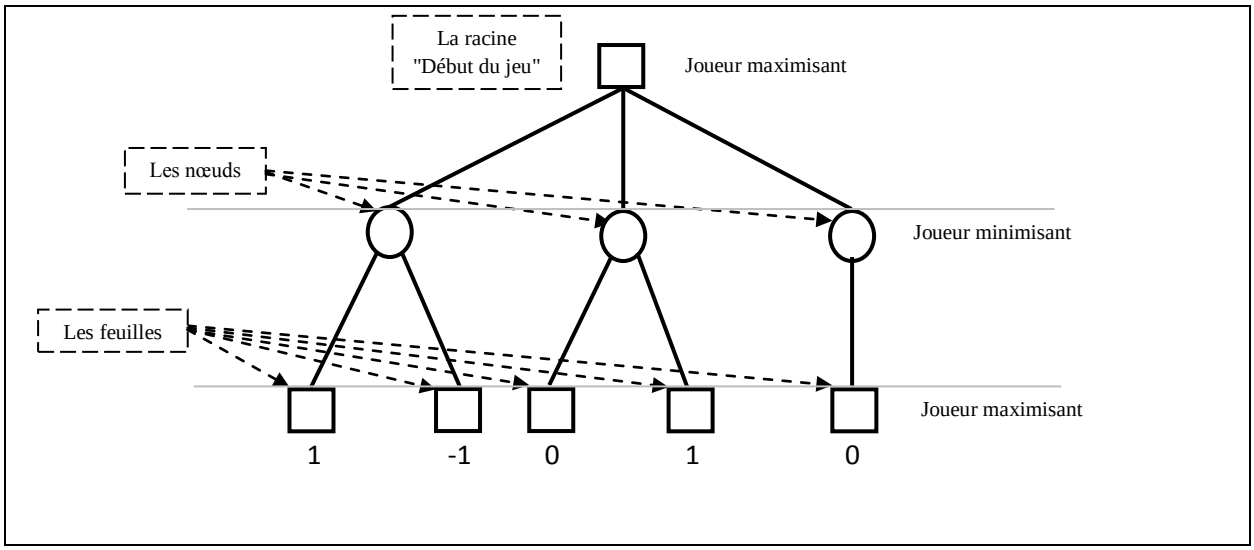


Figure 1.3 : Exemple de représentation d'un arbre de jeu avec des carrés et des cercles.

10. Fonction d'évaluation :

Une fonction d'évaluation est une fonction qui associe à une configuration de jeu et à un joueur donné, une valeur réelle. Cette valeur est sensée estimer la qualité de la configuration de jeu pour le joueur, en termes de ses chances de gagner la partie.

Les première fonctions proposées furent les plus simples. Aux jeux de dames par exemple, on peut effectuer la différence entre le nombre de ses pions et le nombre de pions de l'adversaire. Il est clair qu'une fonction heuristique élaborée permet de mieux jouer « *si j'inclus dans ma fonction d'évaluation des notions d'attaque et de défense de pions, j'améliore mon niveau de jeu* ». [11]

***Remarque* :** Plus la fonction d'évaluation est élaborée, plus il faut de temps pour la calculer, ce qui limitera la profondeur d'exploration de l'arbre.

11. Les algorithmes de recherche :

Pour pouvoir résoudre un problème, un *algorithme de recherche* doit réaliser une exploration de l`espace d`états d`une manière systématique et contrôlée.

11.1. L'algorithme de Minimax :

L'algorithme Minimax peut être utilisé avec la forme normale (matricielle) ou avec la forme extensive (arborescente). Voyons ces deux applications :

11.1.1. Forme normale :

Les joueurs étant prudents, ils vont chercher à *minimiser leurs pertes*. Ils vont donc commencer par chercher ce qui peut leur arriver de pire pour chaque stratégie (ligne/colonne), puis par choisir, parmi ces cas de figure, celui qui leur est le moins défavorable.

- **Joueur maximisant :** Le joueur maximisant prend donc, pour chaque ligne la plus petite valeur algébrique. Il choisit ensuite la stratégie (ligne) correspondant à la plus grande de ces valeurs. Il va donc choisir le maximum parmi les minimums (max min).
- **Joueur minimisant :** De même, le joueur minimisant prend, pour chaque colonne, la plus grande valeur algébrique (ses pertes correspondent aux gains du joueur maximisant). Il choisit ensuite la stratégie (colonne) correspondant à la plus petite de ces valeurs. Il va donc choisir le minimum parmi les maximums (min max).

***Exemple* :** On a ici une matrice de jeu.

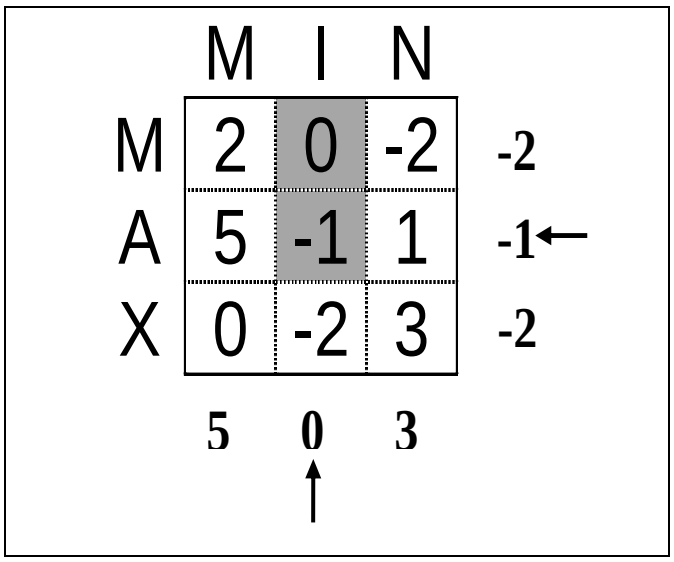


Figure 1.4 : Exemple d'application de l'algorithme Minimax avec la représentation matricielle.

Le joueur maximisant commence par calculer les pires cas pour chaque ligne (stratégies). Il choisit ensuite celle qui lui causera le moins de pertes : la stratégie 2, avec son résultat de 2.

Le joueur minimisant fait de même avec les colonnes (ses stratégies), mais de son point de vue, il cherche donc les plus grandes valeurs. Il choisit ensuite celle qui lui causera le moins de pertes : la stratégie 1 avec son résultat de 2.

***Remarque* :** Lorsque l'on a maxmin = minmax, alors on est en présence d'un *point col*, et réciproquement. Les deux stratégies (celle du joueur maximisant et celle du joueur minimisant) aboutissant au point col sont des stratégies dites *pures optimales* « on ne pourra pas obtenir un résultat plus désavantageux en jouant autre chose ».

On peut avoir plusieurs points col. Mais, dès que l'on a un, on ne devra pas appliquer de stratégies mixtes, puisqu'aucun joueur ne pourra faire mieux.

***Exemple* :** On est ici en présence d'un jeu avec point col.

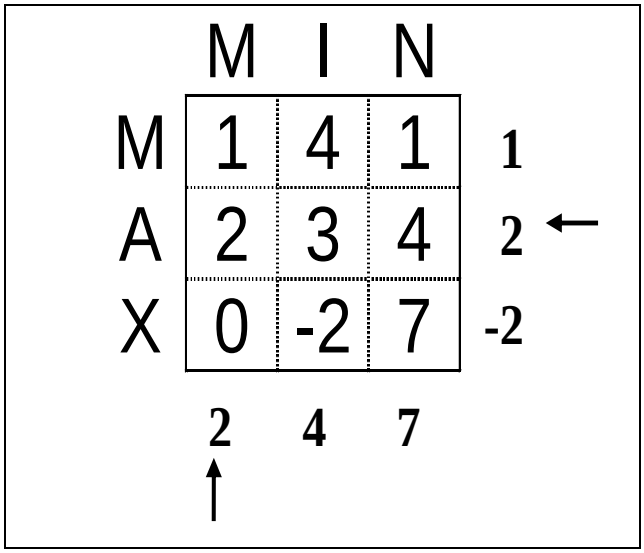


Figure 1.5 : Exemple de représentation matricielle en présence d'un jeu avec point col.

En effet, après avoir appliqué l'algorithme du Minimax, comme dans l'exemple précédent, on s'aperçoit que la valeur issue des deux stratégies optimales est la même : 2.

Si le joueur maximisant utilise une des deux stratégies 1 ou 3, il aura un gain inférieur (1 ou 0) à celui apporté par la stratégie optimale. De même, si le joueur minimisant choisit une stratégie autre que la première, il perdra encore plus (3 ou 4 au lieu de 2). Aucun des deux joueurs ne peut donc faire mieux que son mieux.

11.1.2. Forme extensive :

L'algorithme du Minimax prend en considération le fait que l'adversaire va toujours jouer son meilleur coup. Il est donc nécessaire, pour chaque joueur de rechercher le coup qui va lui assurer le minimum de perte "*stratégie sécurisante*".

Ainsi, à chaque nœud où le joueur maximisant doit prendre une décision, il va considérer la plus grande des valeurs de ses fils. Réciproquement, le joueur minimisant prendra la plus petite des valeurs de ses fils. [10]

***Exemple* :**

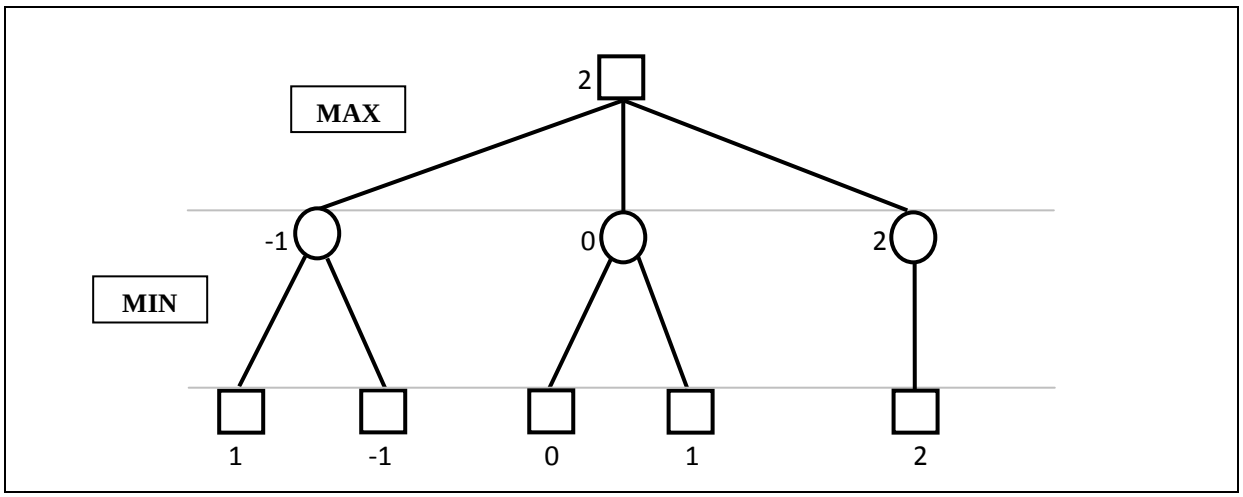


Figure 1.6 : Exemple de calcul de la valeur d'un nœud en fonction des valeurs de ses fils.

Cette notation se fait donc en remontant l'arbre, c'est à dire les feuilles les plus éloignées jusqu'aux descendants de la racine. À ce stade, le premier joueur doit jouer le coup qui maximise la note affectée aux descendants.

L`algorithme Minimax peut être décrit par le pseudo code suivant :

Minimax

```
int fonction minimax (int profondeur)
{
   Si (jeu est terminé ou profondeur = 0)
   retourner Score résultant ou eval();
int Current;
Mouvement MeilleurMouvement;
   Si (nœud == MAX) {
      Current = -INFINI;
      Pour chaque (Mouvement m) {
         Faire le mouvement m;
         int score = minimax (profondeur - 1)
         Retirer le mouvement m;
         Si (score > Current) {
             Current = score;
             MeilleurMouvement = m ;
         }
      }
   }
   Sinon {
      Current = +INFINI;
      Pour chaque (Mouvement m) {
         Faire le mouvement m;
         int score = minimax (profondeur - 1)
         Retirer le mouvement m;
         Si (score < Current) {
             Current = score;
             MeilleurMouvement = m ;
         }
      }
   }
   retourner Current;
}
```

Algorithme 1.1 : Algorithme de Minimax.

***Exemple* :**

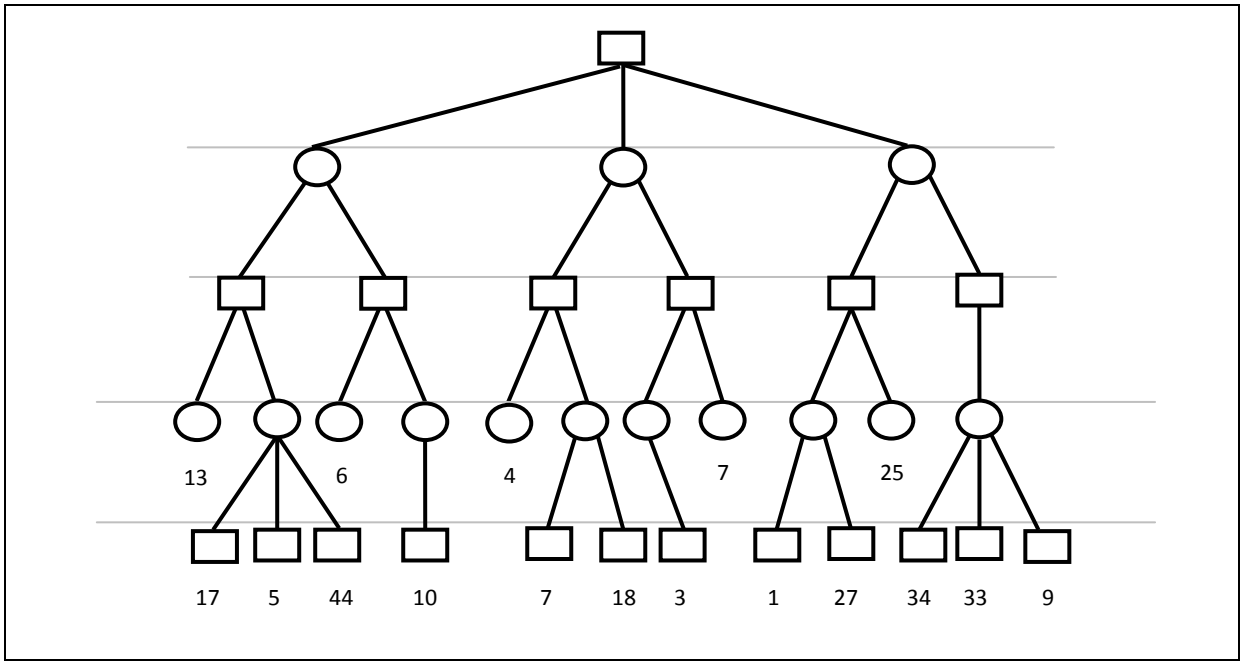


Figure 1.7 : Exemple d'arbre de jeu.

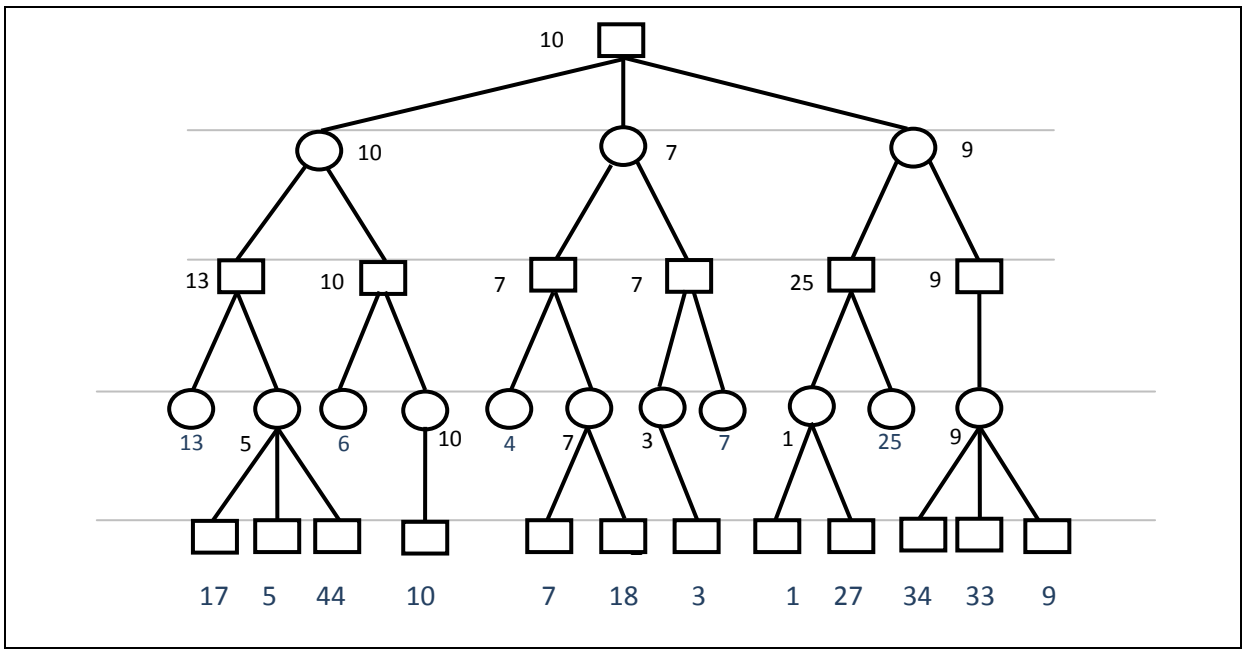


Figure 1.8 : Exemple résolu avec l'algorithme du Minimax.

11.1.3. Les limites de l'algorithme Minimax :

L'algorithme du Minimax est bien entendu, inutilisable dans la pratique, puisqu'il explore toutes les branches de l'arbre sans distinction alors que certaines branches n'apporteront rien. Son principe est néanmoins repris par tous les autres algorithmes dont les améliorations consisteront principalement à introduire des méthodes d'élagage. C'est à dire que l'on va éviter d'analyser les branches de l'arbre qui ne peuvent plus influencer le résultat à un moment donné du calcul.

On observe que l`algorithme Minimax effectue l`évaluation pour tous les nœuds de l`arbre de jeu d`un horizon donné. Mais il existe des situations dans lesquelles, pour déterminer la valeur Minimax associée à la racine, il n'est pas nécessaire de calculer les valeurs associées à tous les nœuds de l'arbre. [10]

11.2. L'élagage Alpha-Bêta :

Une amélioration du Minimax est de faire *un élagage* de certaines branches de l'arbre. L'idée de base est de retenir les valeurs de référence des ancêtres du nœud où l'on se trouve. Ces valeurs servent de référence, permettant ainsi de ne pas parcourir les branches inutiles car ils ne pouvant plus influencer la valeur Minimax de la racine.

En d'autres termes, l'algorithme *alpha-bêta "α–β"* utilise l'histoire de la recherche déjà effectuée lors du début de la recherche minimax pour borner les variations possibles de la notation de la racine, ce qui permet d'élaguer de façon non risquée des sous-arbres entiers de l'espace de recherche (on parle de *coupures alpha-bêta*).

11.2.1. Les limites Alpha et Bêta :

Les deux valeurs de référence conservées sont appelées α et β, d'où le nom de l'algorithme :

- **La limite α :**

La limite α est la limite inférieure de coupure pour un nœud *Min J*. Il représente la valeur courante maximale de tous les ancêtres de *J*. On arrête l'exploration d'un nœud *J* dès que la valeur courante "*cv*" vérifie : $cv \leq \alpha$. On a donc, initialement : $\alpha = -\infty$

- **La limite β :**

La limite β est la limite supérieure de coupure pour un nœud *Max J*. Il représente la valeur courante minimale de tous les ancêtres de *J*. On arrête l'exploration d'un nœud J dès que la valeur courante "*cv*" vérifie : $cv \geq \beta$. On a donc, initialement : $\beta = +\infty$

11.2.2. Avantages par rapport au Minimax :

L'avantage de cet algorithme est qu'il donne exactement les mêmes résultats que l'algorithme Minimax tout en étant beaucoup plus rapide. Si les nœuds sont correctement ordonnés la vitesse de l'algorithme est accrue du fait que les coupes surviennent plus tôt. Ainsi pour un même temps d'exécution, l'algorithme Alpha-Bêta permettra d'aller à une profondeur plus grande.

En effet, Alpha-Bêta est l'algorithme le plus utilisé dans les applications de jeux, en partie grâce à sa consommation de mémoire linéaire. Cette consommation est proportionnelle à la profondeur de l'arbre analysé, puisque l'on stocke à tout moment un couple (α,β) par ancêtre du nœud local.

Bien entendu, par consommation de mémoire, on entend consommation propre à l'algorithme, et non pas la consommation due à l'arbre, celui-ci ne variant pas en fonction de l'algorithme utilisé. [12]

L`algorithme de Alpha-Bêta peut être décrit par le pseudo code suivant :

Alpha-Bêta :

```
int alphabêta(int profondeur, int alpha, int bêta)
{
   Si (jeu est terminé ou profondeur <= 0)
      retourner score résultant ou eval();
   mouvement MeilleurMouvement;
   Si (nœud == MAX) {
      Pour chaque (mouvement m) {
         Faire le mouvement m;
         int score = alphabêta (profondeur - 1, alpha, bêta)
         Retirer le mouvement m;
         Si (score > alpha) {
            alpha = score;
            MeilleurMouvement = m ;
            Si (alpha >= bêta)
               break;
         }
      }
      retourner alpha ;
   }
   Sinon {
      Pour chaque (mouvement m) {
         Faire le mouvement m;
         int score = alphabêta (profondeur - 1, alpha, bêta)
         Retirer le mouvement m;
         Si (score < bêta) {
            bêta = score;
            MeilleurMouvement = m ;
            Si (alpha >= bêta)
               break;
         }
      }
      retourner bêta;
   }
}
```

Algorithme 1.2 : Algorithme d'Alpha-Bêta.

Exemple **:**

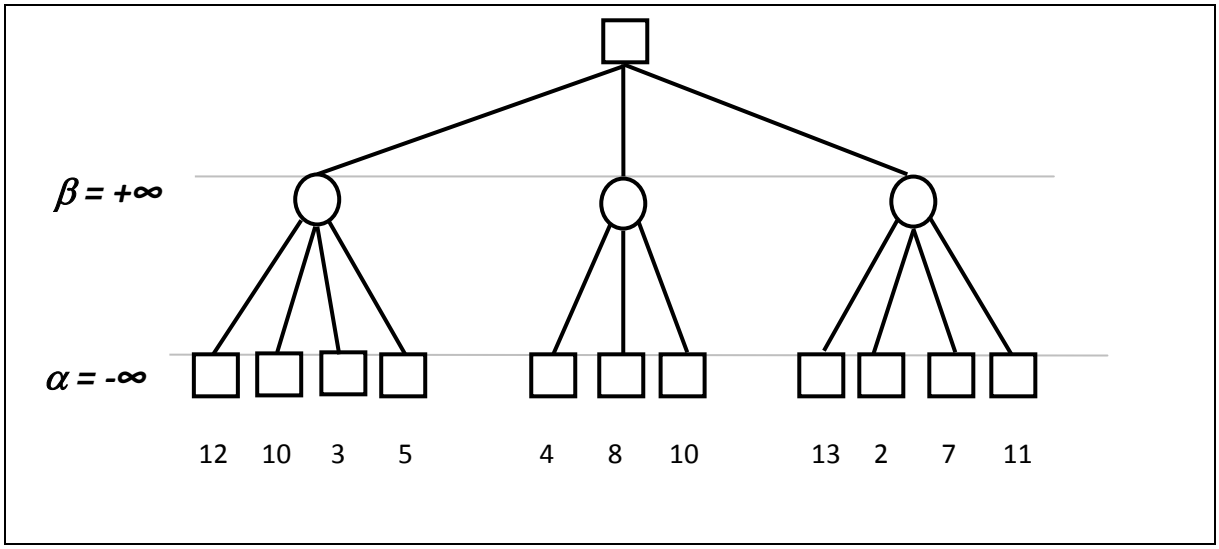


Figure 1.9 : Exemple d'arbre de jeu.

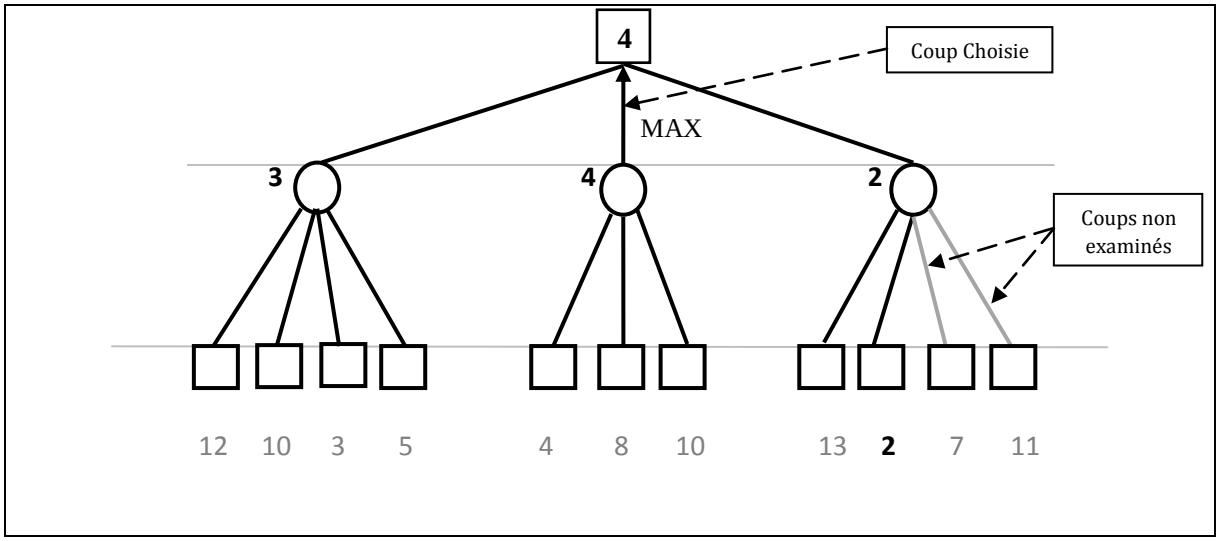


Figure 1.10 : Exemple résolu par l'algorithme α–β Sens de parcours de droite à gauche.

11.3. La convention NegaMAx :

C'est une convention simplificatrice (du moins pour le programmeur) qui consiste à considérer qu'on évalue une position non pas du point de vue d'un joueur fixe (par exemple joueur Max = programme) mais du point de vue du joueur qui a le trait sur cette position.

Pour conserver alpha et bêta pour chaque niveau il suffit de les intervertir entre les appels de procédures et d'inverser leurs valeurs. En d'autres termes, on utilise encore, implicitement, la condition de symétrie des jeux à somme nulle : [13]

Position.eval (Joueur) = - Position.eval (Adversaire)

L`algorithme de Alpha-Bêta, en convention NegaMax peut être décrit par le pseudo code suivant :

Alpha-Bêta, en convention NegaMax

```
int alphabêta(int profondeur, int alpha, int bêta)
{
   Si (jeu est terminé ou profondeur <= 0)
      retourner score resultant ou eval();
   mouvement MeilleurMouvement ;
   Pour chaque (mouvement m) {
      Faire le mouvement m;
      int score = -alphabêta(pronfondeur - 1, -bêta, -alpha)
      Retirer le mouvement m;
      Si (score >= alpha) {
         alpha = score ;
         MeilleurMouvement = m ;
      Si (alpha >= bêta)
         break;
      }
   }
   retourner alpha;
}
```

Algorithme 1.3 : Algorithme d'Alpha-Bêta, en convention NegaMax

11.4. Algorithme Fail-Soft Alpha-Bêta :

L'algorithme de Fail-soft Alpha-Bêta est une amélioration de Alpha-Bêta en modifiant légèrement le pseudo-code de l'algorithme, en convention NegaMax, il est possible de ramener un supplément d'information de la recherche, "*la valeur qui a provoqué la coupure*".

L`algorithme de Fail-Soft alpha-bêta peut être décrit par le pseudo code suivant :

Fail-Soft Alpha-Bêta

```
int alphabêta(int profondeur, int alpha, int bêta)
{
   Si (jeu est terminé ou profondeur <= 0)
      retourner score résultant ou eval();
   mouvement MeilleurMouvement ;
   int current = -INFINI;
   Pour chaque (mouvement m) {
      Faire le mouvement m;
      int score = - alphabêta(profondeur - 1, -bêta, -alpha)
      Retirer le mouvement m;
      Si (score >= current) {
         current = score;
         MeilleurMouvement = m;
         Si (score >= alpha){
            alpha = score;
            MeilleurMouvement = m ;
            Si (score >= bêta)
               break;
         }
      }
   }
   return current;
}
```

Algorithme 1.4 : Algorithme de Fail-Soft Alpha-Bêta

12. Conclusion :

Dans ce chapitre nous avons présenté les notions de base de la théorie des jeux. L'intérêt principal de cette théorie consiste à étudier les différentes interactions entre les individus. La théorie des jeux reste un modèle très fécond, qui suscite de nombreux travaux, en matière de représentation des jeux ainsi que le choix de la meilleure fonction d'évaluation tout en permettant la modélisation de la non-linéarité même sans passer par les équations du modèle mathématique.

CHAPITRE II

ALGORITHMES ÉVOLUTIONNAIRES: PRINCIPES ET MÉTHODES

Ce chapitre présente les principes de l'évolution naturelle et artificielle. Nous détaillons les grandes lignes des algorithmes évolutionnaires et plus particulièrement les algorithmes génétiques et les stratégies d'évolution. Nous présentons ensuite nos choix concernant le type d'évolution que nous allons utiliser dans la suite de ce mémoire.

ALGORITHMES ÉVOLUTIONNAIRES

Les organismes vivants résolvent superbement de nombreux problèmes, et leurs capacités d'adaptation en remontrent aux meilleurs programmes informatiques. De surcroît, les informaticiens passent des mois, voire des années, à construire des algorithmes, alors que les organismes s'élaborent spontanément grâce aux mécanismes de sélection naturelle et d'évolution.

La sélection naturelle élimine l'un des obstacles majeurs à la conception des programmes : la spécificité préalable de toutes les caractéristiques d'un problème et des tâches précises qu'un programme doit effectuer pour résoudre ce problème. En reproduisant informatiquement des mécanismes d'évolution, on « élève » aujourd'hui des programmes qui résolvent des problèmes dont personne ne comprend complètement la structure.

Les ***algorithmes évolutionnaires*** *explorent des espaces de solutions beaucoup plus vastes que les programmes classiques. En outre, l'étude de l'effet de la sélection naturelle sur les programmes, dans des conditions contrôlées et bien comprises, pourrait montrer comment la vie et l'intelligence ont évolué sur la terre.*

La plupart des organismes évoluent par deux mécanismes principaux : la ***sélection naturelle*** *et la* ***reproduction sexuée****. La sélection naturelle détermine quels membres d'une population survient et se reproduisent, la reproduction sexuée assure le brassage et la recombinaison des gènes parentaux, pour former des descendants aux potentialités nouvelles. Lorsqu'un spermatozoïde et un ovule fusionnent, leurs chromosomes s'apparient et s'échangent des segments. Ce mélange permet une évolution beaucoup plus rapide qui si les individus recevaient tous leurs gènes d'un même parent et que si les mutations étaient les seules causes de variations génétiques (bien que les organismes unicellulaire ne s'accouplent pas comme les animaux, ils échangent également du matériel génétique, et leur évolution peut se décrire de la même façon).* [14]

John Holland

1. Introduction :

Il existe une catégorie de problèmes pour lesquels il est difficile, voire impossible, de trouver une solution en un temps limité. Il est alors utile de trouver une technique permettant la localisation rapide de solutions sous-optimales, sachant que l'*espace de recherche* a une *taille* et une *complexité* suffisamment importantes pour éliminer toute garantie d'*optimalité*. Pour cela, un système qui est capable de s'auto-modifier au cours du temps, tout en améliorant sa *performance* dans l'accomplissement des tâches auxquelles il est confronté, semble ouvrir la voie à une recherche intéressante. [15]

L'évolution biologique a engendré des systèmes vivants extrêmement complexes. Elle est le fruit d'une altération progressive et continue des êtres vivants au cours des générations. [4]

Une voie de recherche qui est très active de nos jours est celle qui essaye de développer de nouvelles approches s'inspirant des principes de cette évolution pour traiter plus efficacement les différents problèmes, notamment ceux portant sur l'*optimisation*. [16]

« *La vie est une compétition, et seuls les mieux adaptés survient et se reproduisent* », dit *Darwin*. Cette règle, qui a engendré les organismes que nous connaissons aujourd'hui, est utile pour la résolution de problèmes par ordinateur. [17]

Les principes de la ***sélection naturelle***[1] et de la ***reproduction***, présentés pour la première fois par *Darwin*, ont inspiré bien plus tard les chercheurs en informatique. Ils ont donné naissance à une classe d'algorithmes regroupés sous le nom générique d'***Algorithmes Evolutionnaires*** (Evolutionary Algorithms (EA)). [18]

On distingue quatre grandes familles historiques d'algorithmes évolutionnaires et les différences entre elles ont laissé des traces dans le paysage évolutionnaire actuel, en dépit d'une unification de nombreux concepts.

- Algorithmes Génétiques (*GA*) : J. Holland, 1975 et D.E. Goldberg, 1989. Michigan, USA.
- Stratégies d'évolution (*ES*) : I. Rechenberg et H.P. Schwefel, 1965. Berlin.
- Programmation évolutionnaire (*EP*) : L.J. Fogel, 1966 et D.B. Fogel, 1991, 1995. Californie, USA.
- Programmation génétique (*GP*) : J. Koza, 1990. Californie, USA. [11]

[1] Sélection naturelle. Les gènes les plus performants ont tendance à se diffuser dans la population tandis que ceux qui ont une performance relative plus faible ont tendance à disparaître.

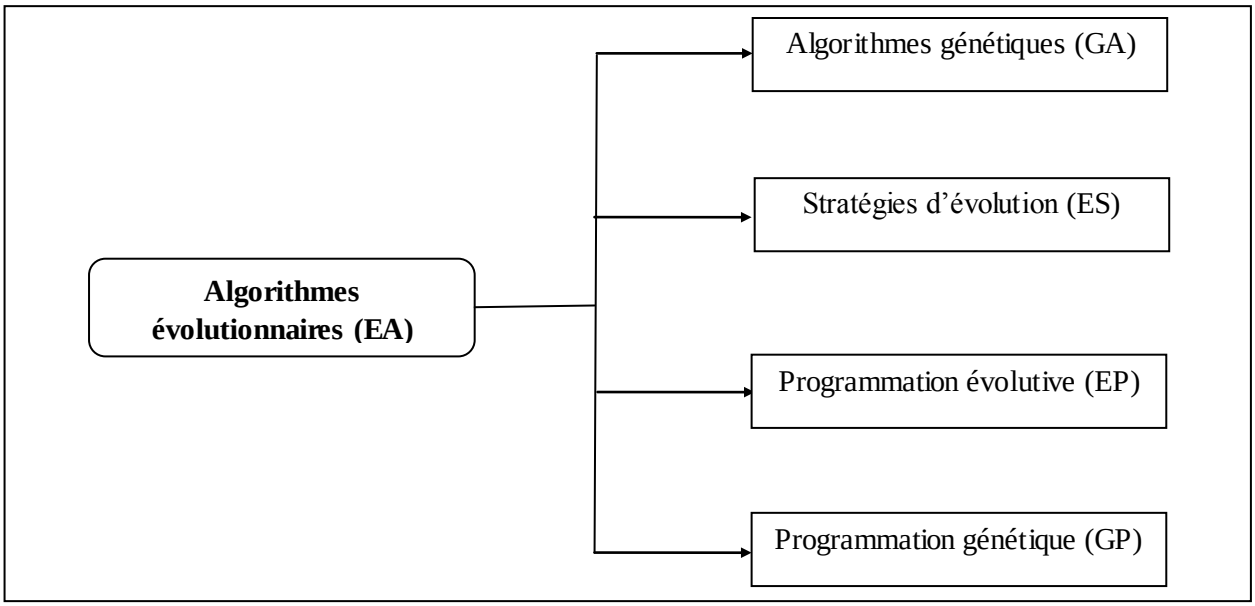


Figure 2.1: Différentes branches des algorithmes évolutionnaires.

2. Les Algorithmes évolutionnaires :

Les *EA* désignent un ensemble d'algorithmes stochastiques qui utilisent les principes de l'évolution naturelle pour résoudre des problèmes divers, très majoritairement d'*optimisation*. Leur fonctionnement repose sur la *génération aléatoire* de solutions potentielles, puis la *sélection* des meilleurs candidats selon un critère préétabli [19]. Ils commencent avec une population initiale, candidate, qui évolue vers la solution optimale ou du moins vers une solution proche de l'optimale.

Ces algorithmes résolvent le problème de la *recherche* des bons éléments, les meilleurs produisent plus souvent que les mauvais. En d'autres termes, si tout va bien dans l'algorithme les enfants doivent être meilleurs que les parents. [20]

Les *EA* ont pour but d'optimiser une fonction réelle. En effet, peu de connaissances sur la manière de résoudre ces problèmes sont nécessaires, même si certaines peuvent être exploitées afin de rendre plus efficace l'évolution (en effet, il n'est pas réaliste d'espérer obtenir une méthode d'optimisation raisonnablement efficace si aucune connaissance sur le domaine à traiter). C'est pourquoi, dans de nombreux domaines, les chercheurs ont été amenés à s'y intéresser. [21]

Un EA est défini par :

- **Séquence[1]/Individu/Chromosome :** une solution potentielle du problème qui correspond à une valeur codée de la variable (ou des variables) en considération.
- **Population :** un ensemble de chromosomes ou de points de l'espace de recherche (donc des valeurs codées des variables).
- **Espace de recherche :** l'environnement (caractérisé en termes de performance correspondant à chaque individu possible).
- **Fonction de performance (fitness) :** appelée aussi fonction d'évaluation, c'est la fonction - positive - que nous cherchons à maximiser car elle représente l'adaptation de l'individu à son environnement. [22]

3. Description détaillée des algorithmes évolutionnaires :

Voici une description plus précise d'un *EA* :

Il s'agit d'un algorithme itératif de recherche globale dont le but est d'*optimiser la fonction de performance* définie par l'utilisateur, pour atteindre cet objectif, l'algorithme travaille en parallèle sur une population d'*individus* (chromosomes[2]), distribués dans l'entièreté de *l'espace de recherche* (l'environnement).

La fonction de performance d'un individu est normalement indépendante des autres individus de la population (elle est explicitement donnée par l'utilisateur). Chaque individu ou chromosome est constitué d'un ensemble d'éléments appelés *caractéristiques* ou *gènes*[3], pouvant prendre plusieurs valeurs appelées *allèles* appartenant à un alphabet non nécessairement numérique. Dans l'algorithme de base, les allèles possibles sont 0 et 1, et un chromosome est donc une chaine binaire [23]. *Exemple* : 001001

Le but est de chercher la *combinaison optimale* de ces éléments, qui donne lieu au maximum de performance. A chaque itération, appelé *génération*, est créée une nouvelle population avec le même nombre d'individus.

Cette nouvelle génération consiste généralement en des individus mieux *adaptés* à l'environnement tel qu'il est présenté par la fonction de performance. La génération d'une nouvelle population à partir de la précédente s'effectue en trois étapes : *Evaluation*, *Sélection* et *Reproduction*.

[1] Séquence. Une séquence A de longueur L(A) une séquence A= {a1, a2, …, a $_i$} avec ∀ i∈ {1,…, L}, a_i∈ V= {0,1}.

[2] Chromosomes. Une suite de gène.

[3]Gène. Un gène est repérable par sa position (son locus) sur le chromosome en question.

4. Principes généraux des algorithmes évolutionnaires :

Les *EA* sont une classe d'algorithmes d'optimisation par recherche probabiliste. Ils modélisent une population d'individus par des points dans un espace [21].

Quel que soit le type de problème à résoudre, les EA opèrent selon les principes suivants :

- La population est *initialisée* de façon dépendante du problème à résoudre (l'environnement).
- La population *évolue* de génération en génération à l'aide d'opérateurs de sélection, de recombinaison et de mutation.
- L'environnement a pour charge d'*évaluer* les individus en leur attribuant une performance. Cette valeur favorisera la *sélection des meilleurs individus*, en vue, après reproduction (opérée par la mutation et/ou recombinaison), d'améliorer les performances globales de la population. [24]

La Figure suivante présente l'organigramme d'un AE :

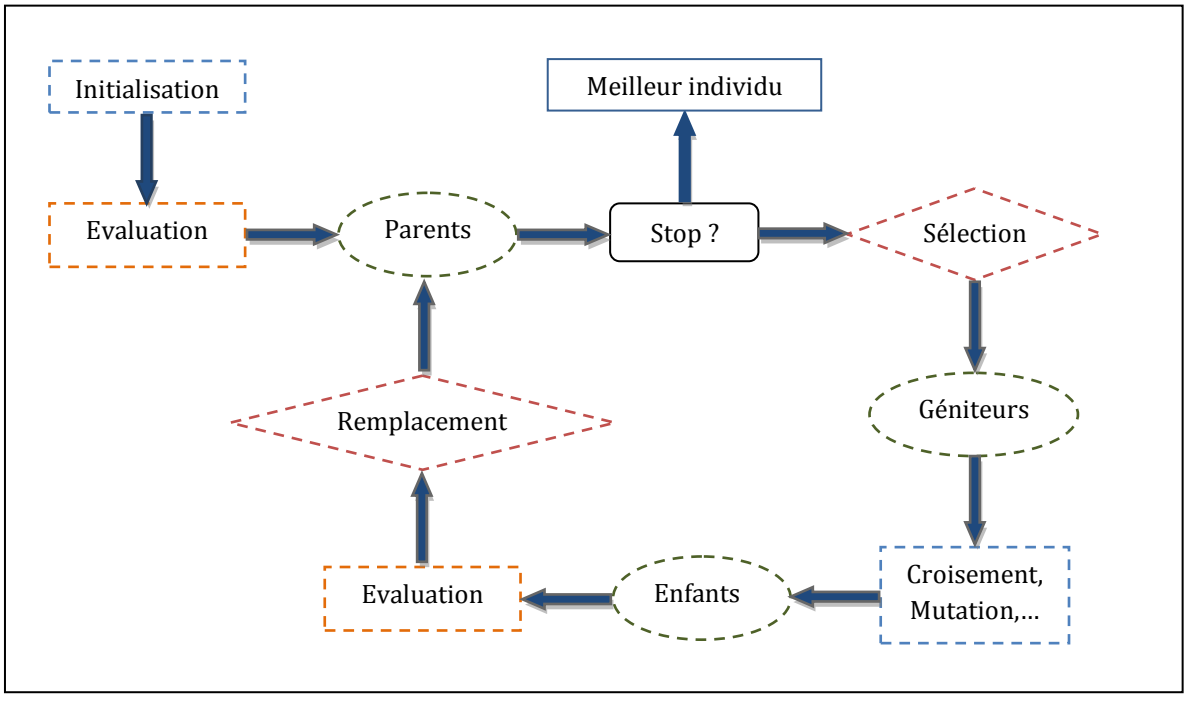


Figure 2.2: Organigramme d'un algorithme évolutionnaire.

5. Éléments de base d'un Algorithme évolutionnaire :

Pour utiliser un EA, on doit disposer des cinq éléments suivants :

5.1. Un principe de codage de l'élément de population :

Cette étape associe à chacun des points de l'espace d'état une structure de données. Elle se place généralement après une phase de modélisation mathématique du problème traité. La qualité du codage des données conditionne le succès des algorithmes évolutionnaires.

Les *codages binaires* ont été très utilisés à l'origine, ils ont donné de très bons résultats. Cependant, de nouvelles versions modifiées des algorithmes évolutionnaires originaux ne se basent plus sur le codage binaire des paramètres à optimiser, mais travaille directement sur les paramètres eux-mêmes, on parle alors des codages réels. [25]

Les *codages réels* ont été utilisés dans des systèmes de commande, dans l'apprentissage des réseaux de neurones, le chromosome dans ce cas est une chaîne de réels. [20]

5.2. Un mécanisme de génération de la population initiale :

Ce mécanisme doit être capable de produire une population d'individus non homogène qui servira de base pour les générations futures. Dans le cas où l'on ne connaît rien du problème à résoudre, il est essentiel que la population initiale soit répartie sur tout le domaine de recherche.

5.3. Une fonction à optimiser :

Celle-ci retourne une valeur appelée *fonction de performance* ou d'évaluation de l'individu. Elle doit traduire correctement le problème à résoudre c'est-à-dire que l'évaluation des chromosomes qui sont les solutions candidate au problème posé doit être pertinente [20]. Si par exemple on avait à chercher le maximum d'une fonction (problème posé), un chromosome devrait représenter les valeurs des variables de la fonction, et une bonne fonction d'évaluation serait la fonction même à maximiser. [25]

5.4. Des opérateurs :

Permettant de diversifier la population au cours des générations et d'explorer l'espace d'état.

5.4.1. Opérateur de sélection :

L'algorithme génétique sélectionne les individus sur la base de leur fonction de performance, plus précisément, l'opérateur de sélection sélectionne chaque individu *i* avec une probabilité $\boldsymbol{fi/\sum fj}$ (la somme étant prise sur les *n* individus constituant la population), comme l'opérateur de sélection est appliqué n fois, l'espérance mathématique du nombre d'enfants de *i* sera $\boldsymbol{n\ fi/\sum fj}$. Les individus sélectionnés constituent une nouvelle population. [26]

Les différentes méthodes de sélection :

5.4.1.1. Sélection par roulette :

Les parents sont sélectionnés en fonction de leur performance. Lors du tirage d'un individu à la roulette, un individu *S* est sélectionné selon une probabilité proportionnelle à *S*.

5.4.1.2. Sélection par rang :

La sélection par rang trie d'abord la population par la fonction d'évaluation. Ensuite, chaque individu se voit associé un rang en fonction de sa position. L'individu le plus mauvais aura le premier rang, le suivant aura le deuxième rang, et ainsi de suite jusqu'au meilleur individu qui aura le rang *N* (pour une population de *N* individu).

5.4.1.3. Sélection par tournois :

Choix uniforme de deux individus dans une population, puis choix du meilleur de l'échantillon, c'est-à-dire sur une population de *N* chromosomes, on forme *N/2* paires chromosomes ensuite on sélectionne le meilleur de chaque paire. Dans ce type de sélection, on détermine une *probabilité de victoire* du plus fort, cette probabilité représente la chance que le meilleur chromosome de chaque paire soit sélectionné.

5.4.1.4. Elitisme :

À la création d'une nouvelle population, il y a une grande chance que les meilleurs chromosomes soient perdus après les opérations d'hybridation et de mutation. Pour éviter cela, on utilise la méthode d'élitisme. Elle consiste à copier un ou plusieurs des meilleurs chromosomes dans la nouvelle génération. Ensuite, on génère le reste de la population selon l'algorithme de reproduction. Cette méthode améliore considérablement les algorithmes évolutionnaires, car elle permet de ne pas perdre les meilleures solutions.

5.4.2. Opérateur de mutation :

La mutation agit en modifiant aléatoirement un ou plusieurs gènes (bits) d'un chromosome. Cet opérateur est appliqué avec une probabilité fixée (*Pm*) sur chacun des individus de la population.

L'opérateur de mutation fonctionne comme suit :

Mutation

```
Pour chaque bit de l'individu
  Générer un réel aléatoire r, tel que r ∈ [0, 1]
  Si r < Pm alors
    le bit sera inversé
```

Algorithme 2.1 : Algorithme de mutation.

5.4.3. Opérateur de croisement :

Le croisement permet de créer de nouvelles chaines en échangeant de l'information entre deux chaines. Le croisement s'effectue en deux étapes :

Etape 01 **:** les nouveaux éléments produits par la reproduction appariés.

Etape 02 **:** chaque paire de chaines subit un croisement comme suit : un entier *k* représentant une position sur la chaine est choisi aléatoirement dans l'intervalle [1, (L – (L-1))], tel que *L* est la langueur de la chaine. Deux nouvelle chaine sont crées en échangeant tous les caractères compris entre les positions *k+1* et *L* inclusivement.

Le croisement peut se faire selon deux méthodes :

5.4.3.1. Croisement en deux points :

On choisi au hasard deux point de croisement et on échange les parties de chaine situées entre ces deux points. [26]

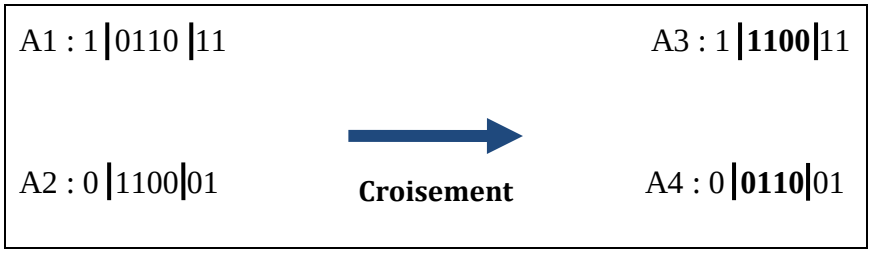


Figure 2.3 : Croisement en deux points

5.4.3.2. Croisement uniforme (multi-points) :

Dans ce type de croisement, on utilise un *masque de croisement*, qui consiste en un vecteur généré aléatoirement, de longueur identique aux chaines parents, et composé de 0 et 1.

Lorsque le bit du masque vaut 0, l'enfant hérite le bit du premier parent, sinon il hérite du second parent. Le seconde enfant est le complémentaire du premier. Ce croisement peut être considéré comme une génération du croisement *multipoints* sans connaissance préalable du point de croisement. [26]

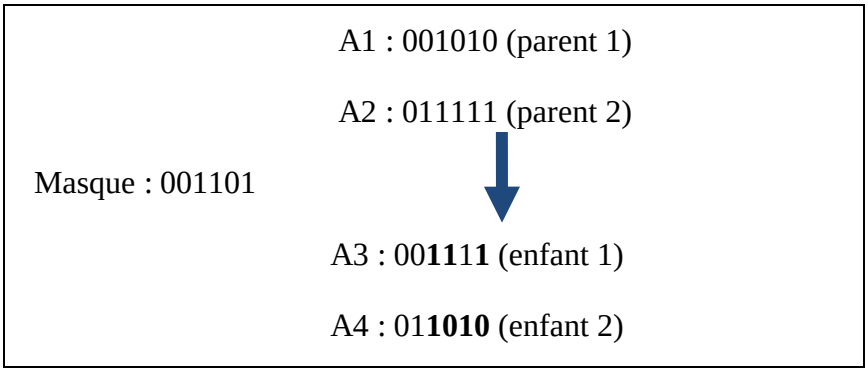


Figure 2.4 : Croisement uniforme.

5.5. Des paramètres de dimensionnement :

La *taille de la population*, le *nombre total de générations* ou *critère d'arrêt*, les *probabilités* d'application des opérateurs de croisement et de mutation.

Dans les sections suivantes, nous allons présenter deux méthodes classiques des *EA* qui sont les algorithmes génétiques (*GA*) et les stratégies d'évolution (*ES*) afin de choisir une des deux méthodes pour la réalisation de notre mémoire.

6. Algorithmes génétiques :

Développés dans les années 70 avec le travail de Holland puis approfondis par Goldberg, Les *GA* sont des algorithmes d'optimisation stochastique fondés sur les mécanismes de la *sélection naturelle* et de la *génétique* [4]. Ils s'inspirent des mécanismes de l'évolution biologique pour les transposer à la recherche de solutions adaptées au problème qu'on cherche à résoudre.

Ce sont des algorithmes *robustes* car ils peuvent résoudre des problèmes fortement non-linéaires et discontinus, et *efficaces* car ils font évoluer non pas une solution mais toute une population de solutions potentielles et donc ils bénéficient d'un parallélisme puissant.

Les *GA* sont certainement la branche des *EA* la plus connue et la plus utilisée. La particularité de ces algorithmes est le fait qu'ils font évoluer des populations d'individus codés par une chaîne binaire. Ils utilisent les opérateurs de mutation binaire et de recombinaison de différents types. [22]

6.1. Principes de fonctionnement :

A partir d'un problème qu'on cherche à résoudre, Le fonctionnement d'un *GA* est défini par les phases suivantes :

- **Initialisation:** Une population initiale de N chromosomes est tirée aléatoirement.
- **Évaluation:** Chaque chromosome est décodé, puis évalué.
- **Sélection:** Création d'une nouvelle population de N chromosomes par l'utilisation d'une méthode de sélection appropriée.
- **Reproduction:** Possibilité de croisement et de mutation au sein de la nouvelle population.
- **Retour:** A la phase d'évaluation tant que la condition d'arrêt du problème n'est pas satisfaite.

On va détailler ces principes dans ce qui suit :

6.1.1. Sélection :

La sélection proposée par Goldberg consiste à sélectionner les individus proportionnellement à leur performance. Un individu ayant une forte valeur d'adaptation a alors plus de chances d'être sélectionné qu'un individu mal adapté à l'environnement. [4]

6.1.2. Croisement :

Le croisement (recombinaison) consiste à sélectionner aléatoirement une position de césure et de permuter les parties droites des deux parents. [4]

***Exemple illustratif* :**

Un entier *k* est choisi aléatoirement entre 1 et la taille *L* des chaînes moins 1.

Les nouvelles chaînes sont créées en échangeant tous les caractères compris entre les positions *k+1* et L inclues.

Par exemples, considérons les deux chaînes binaires *A* et *B* (parents) et supposons k=2, on obtient alors les chaînes *A1* et *B1* (fils):

Parents		Fils
A : 00 | 101	Recombinaison →	A1 : **01**101
B : 01 | 011		B1 : **00**011

Figure 2.5 : Application de l'opérateur de Croissement

6.1.3. Mutation :

Une mutation consiste simplement en l'inversion d'un bit (ou de plusieurs bits, mais vu la probabilité de mutation c'est extrêmement rare) se trouvant en un locus bien particulier et lui aussi déterminé de manière aléatoire. [4]

***Exemple illustratif* :**

La mutation consiste juste à choisir aléatoirement un caractère d'une chaîne et à le modifier.

Par exemple, soit les chaînes binaires suivantes :

Chromosomes avant mutation		Chromosomes après mutation
0**0**101	Mutation →	0**1**101
11**1**00		11**0**00

Figure 2.6 : Application de l'opérateur de Mutation

6.1.4. Remplacement :

Le remplacement consiste à réintroduire les descendants obtenus par application successive des opérateurs de sélection, de croisement et de mutation (la population P') dans la population de leurs parents (la population P).

Un GA générique à la forme suivante :

Algorithme Génétique :

```
1. Initialiser la population initiale P.
2. Evaluer P.
3. TantQue (Pas Convergence) faire :
     a. P ' = Sélection des Parents dans P ;
     b. P ' = Appliquer Opérateur de Croisement sur P ' ;
     c. P ' = Appliquer Opérateur de Mutation sur P ' ;
     d. P = Remplacer les Anciens de P par leurs Descendants
        de P ' ;
     e. Evaluer P ;
   FinTantQue
```

Algorithme 2.2 : Algorithme génétique.

***Remarque*:** Le critère de convergence peut être de nature diverse, par exemple :

- Un taux minimum qu'on désire atteindre d'adaptation de la population au problème.
- Un certain temps de calcul à ne pas dépasser.
- Une combinaison de ces deux points.

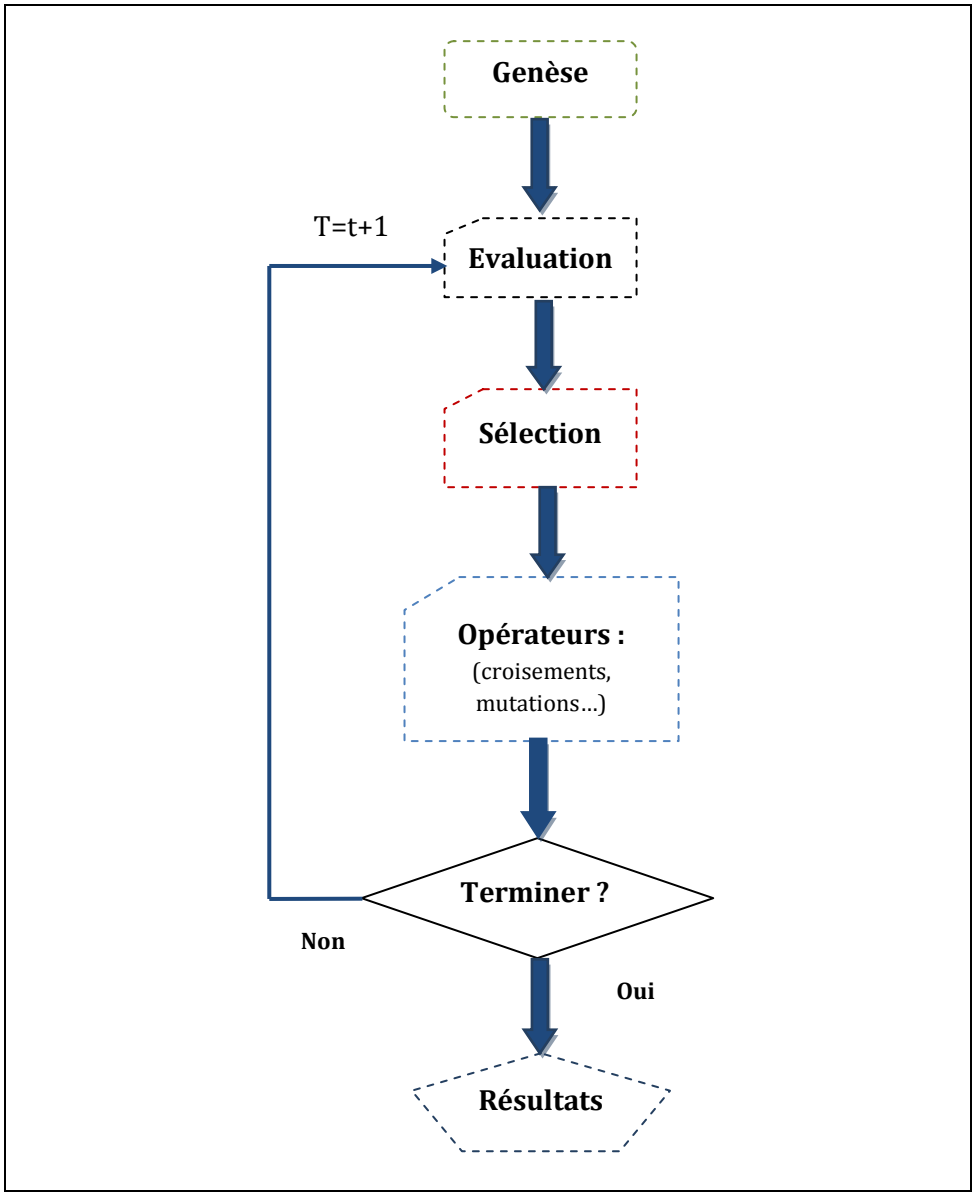


Figure 2.7: Organigramme d'un algorithme génétique.

7. Stratégies d'évolution :

Les *ES* sont apparues dans les années 70 avec les travaux de Ingo Rechenberg, ensuite ces travaux ont été poursuivis par Hans-Paul Schwefel. [27]

La première particularité de ces méthodes est de coder les paramètres du problème à résoudre en nombres réels. La seconde est d'effectuer une sélection déterministe des individus en ne choisissant que les *n* individus classés selon leur performance. La troisième, enfin, est d'encoder les paramètres d'évolution directement dans le génotype afin de les faire évoluer au même titre que les valeurs des paramètres solutions du problème. [4]

Les ES favorisent la mutation plutôt que la recombinaison. Travaillant sur des réels, la mutation suit une loi généralement gaussienne avec des écarts-types généralement codés dans le génotype.

7.1. Principes de fonctionnement :

Le fonctionnement des ES est défini comme suit :

7.1.1. Sélection :

La sélection des individus est déterministe. Deux types de sélection existent, qui sont les sélections (μ, λ) et $(\mu+\lambda)$.

- La première (μ, λ) : consiste à sélectionner les μ meilleurs parmi les λ enfants.
- La seconde $(\mu+\lambda)$: sélectionne les μ meilleurs individus parmi les μ parents de la génération précédente et les λ enfants créés (chaque parent créant λ/μ enfants avec $\lambda>\mu$).

Cette dernière méthode permet de ne pas perdre les meilleurs individus d'une génération à une autre mais accroît les possibilités que la population converge prématurément vers une solution qui ne peut pas être optimale mais qui représente un minimum local.

7.1.2. Recombinaison :

La recombinaison opère ici rarement sur le génotype contenant les variables du problème. Cependant, elle semble très utile pour l'évolution des paramètres de mutation.

7.1.3. Mutation :

Le codage étant réel, le problème se pose quant à la réalisation de la mutation. Les stratégies d'évolution proposent d'utiliser un modèle basé sur des distributions normales, avec des écarts-types qui peuvent être directement codés dans le génotype.

En considérant la représentation simplifiée (sans direction d'évolution) définie plus haut, la mutation est alors : $\forall i \in \{1, \ldots, k\}$ **indice des gènes de** $\vec{x}$ **et** $\vec{\sigma}$:

$$\begin{cases} \sigma i' = \sigma i \,.\, Exp\,(\tau' \,.\, N(0,1) + \tau \,.\, Ni(0,1)) \\ xi' = xi + \sigma i' \,.\, Ni(0,1) \end{cases}$$

Ni(0, 1): représente la réalisation d'une variable suivant une loi normale d'espérance 0 et d'écart type 1 calculée pour chaque indice i.

N(0, 1): une variable de même type calculée une seule fois par individu.

τ et τ': considérés comme des taux d'apprentissage.

σ: une variable aléatoire gaussienne centrée en zéro et d'écart-type σ (ajustée au cours de l'évolution).

Hans-Paul Schwefel propose pour des résultats robustes l'initialisation des paramètres τ et τ' suivante :

$$\begin{cases} \tau' = 1/SQRT\,(2K)\ ; \\ \tau = 1/SQRT\,(2SQRT\,(K))\ ; \end{cases}$$

8. Comparaison entre les GA et les ES : [22]

- **Les algorithmes génétiques :**

Ce sont les algorithmes évolutionnaires les plus usuels. Les génotypes évolués sont des vecteurs de valeur (souvent binaires), qui correspondent à un phénotype et dont on évalue la capacité à résoudre un problème. A chaque génération, tous les parents disparaissent et laissent place à une population totalement composée de leurs descendants.

- **Les stratégies d'évolution :**

Elles sont très similaires à un algorithme génétique, mais cette fois la population est recréée à partir des parents et des enfants. Autrement dit, la sélection se base sur un classement regroupant parents et enfants. Ces derniers ne sont sélectionnés que s'ils sont mieux adaptés

que leurs parents. Les ES utilisent directement un vecteur de nombres réels. Le croisement n'est pas utilisé dans les premières versions, mais des variantes plus élaborées inspirées des AG l'utilisent.

9. Les Algorithmes évolutionnaire et la théorie des jeux :

Dans le domaine de la théorie des jeux, l'exemple le plus connu de l'utilisation des algorithmes évolutionnaires (*EA*) pour résoudre un problème standard est le travail d'*Axelrod* sur l'émergence de la coopération dans le jeu du dilemme du prisonnier répété.

Axelrod (1987) utilise les *EA* pour faire évoluer une population de stratégies qui jouent au dilemme du prisonnier répété à deux joueurs contre toutes les autres stratégies dans la population. [50]

Dans cette approche, chaque chromosome représente l'histoire récente des choix et des observations de chaque joueur. La performance de chaque stratégie est alors évaluée dans ce jeu. L'environnement de chaque stratégie est formé par la population des autres stratégies dans la population. Comme cette population évolue dans le temps, l'environnement de chaque stratégie évolue aussi.

10. Conclusion :

Dans ce chapitre, nous avons présenté un aperçu sur deux méthodes classiques des *EA*. Nous en avons présenté les principes généraux afin d'en étudier les caractéristiques. Dans le chapitre suivant on va aborder les principes des réseaux de neurones. Nous nous intéressons à entrainer un réseau de neurones par un algorithme évolutionnaire. Nous présentons leur méthode d'apprentissage ainsi que l'impact de l'initialisation de leurs poids sur leur temps de convergence et leurs performances.

« La principale conclusion est que les AG ont déjà dépassé le cadre seul de leur applications techniques aux systèmes artificiels, ils sont prêts à entrer dans le monde réel des systèmes vivants et ils ont même déjà fait leurs premiers pas dans cette direction ».

Kosorukoff et Goldberg (2001)

CHAPITRE III

INTRODUCTION AUX RÉSEAUX DE NEURONES

Ce chapitre présente les principes de fonctionnement des réseaux de neurones. Nous présentons leurs méthodes d'apprentissage basées sur la rétro-propagation du gradient ainsi que l'impact de l'initialisation de leurs poids sur leur temps de convergence et leurs performances.

RÉSEAUX DE NEURONES

On pense communément que « l'ordinateur de l'avenir » sera massivement parallèle et tolérera les erreurs. Toutefois la conception d'une telle machine s'étant avérée étonnamment difficile, nous aurions abandonné depuis longtemps si le cerveau n'était pas une preuve vivante que le traitement parallèle et tolérant les erreurs est possible et très efficace. [28]

John S. Denke

1. Introduction :

Reproduire l'intelligence de l'être humain constitue sans aucun doute le rêve le plus passionnant de beaucoup de chercheurs de notre siècle. Les réseaux de neurones sont fondés sur des modèles qui tentent d'expliquer comment les cellules du cerveau et leurs interconnexions parviennent, d'un point de vue global, à exécuter des calculs complexes.

Au cours des deux dernières décennies, les réseaux de neurones ont constaté un développement fulgurant [29]. Cet intérêt a démarré avec l'application réussie de cette technique puissante pour des problématiques très différentes, et dans des domaines aussi divers que la finance, la médecine, la production industrielle, la géologie ou encore la physique.

Le succès croissant des réseaux de neurones sur la plupart des autres techniques statistiques peut s'attribuer à leur *puissance*, leur *polyvalence* et à leur *simplicité d'utilisation*. Ils sont des techniques extrêmement sophistiquées de modélisation et de prévision, en mesure de modéliser des relations entre des données ou des fonctions particulièrement complexes. [29]

Nous donnerons dans ce chapitre les notions de base pour la compréhension des réseaux de neurones.

2. Historique :

En 1943, le premier neurone formel a été proposé par les deux biophysiciens, *Mc Culloch* et *Pitts*. Leur but était de comprendre les propriétés des systèmes nerveux à partir de composants élémentaires. [30]

Grâce à des modèles à base neurones simplifiés *"Neurones formels"*, ils montrent qu'il est possible de construire des systèmes vérifiant la définition de Turing pour les machines à calculer à usage général et donc capables de calculer des fonctions logiques. [31]

En 1949, *Donald Hebb* s'attaque au problème de l'apprentissage, il a proposé une formulation du mécanisme d'apprentissage, sous la forme d'une règle de modification des connexions synaptiques "*règle de Hebb*". Cette règle décrit la manière dont les cellules apprennent à modifier l'intensité des connexions qui les relient. [30]

En 1956, a lieu une grande conférence à Datrmouth sur le thème de l'intelligence artificielle et de l'apprentissage, elle sera le point de départ de l'âge d'or des réseaux de neurones et de l'intelligence artificielle. [31]

En 1958, l'apparition du premier réseau de neurones artificiel, grâce aux travaux de *Rosenblatt* qui conçoit le fameux *"perceptron"*. Le perceptron est inspiré du système visuel (en termes d'architecture neurobiologique) et possède une couche de neurones d'entrée (perceptive) ainsi qu'une couche de neurones de sortie (décisionnelle). [30]

En 1965, *Nilsson* publie "Machine Learning" qui donne les fondements mathématiques de l'apprentissage automatique pour la reconnaissance des formes. [31]

En 1969, une critique violente du perceptron par *Minsky* et *Papert* qui montrent dans un livre "*Perceptrons*" toutes les limites de ce modèle, et soulèvent particulièrement l'incapacité du perceptron à résoudre les problèmes non linéairement séparables, tels que le célèbre problème du XOR (ou exclusif), et les difficultés théoriques posées par l'apprentissage dans les réseaux multicouches. [30]

En 1982, *Hopfield* a démontré tout l'intérêt de l'utilisation des réseaux récurrents "*feed-back*" pour la compréhension et la modélisation des processus mnésiques. Les réseaux récurrents constituent alors la deuxième grande classe de réseaux de neurones, avec les réseaux type perceptron "*feed-forward*". [31]

En 1986, *Werbos* conçoit son algorithme de *rétro propagation* de l'erreur, qui offre un mécanisme d'apprentissage pour les réseaux multicouches de type perceptron appelés *MLP* (Multi Layer Perceptron), fournissant ainsi un moyen simple d'entraîner les neurones des couches cachées. [31]

3. Inspiration biologique :

Le cerveau humain est en réalité constitué d'un très grand nombre de neurones (de l'ordre de cent milliards), étroitement connectés entre eux par plusieurs milliers d'interconnexions pour chaque neurone. Chaque neurone est une cellule spécialisée, capable de *créer*, *envoyer* et *recevoir des signaux électrochimiques*.

Comme toutes les cellules biologiques, les neurones possèdent un *corps cellulaire*, des prolongements apportant des informations au neurone "*les dendrites*", et un prolongement qui communique les informations recueillies par le neurone "*les axones*". L'axone d'une cellule est connecté aux dendrites d'une autre par l'intermédiaire d'une "*synapse*".

Lorsqu'un neurone est activé, il envoie un signal électrochimique au travers de l'axone. Cette impulsion traverse les synapses vers des milliers d'autres neurones, qui peuvent à leur tour, envoyer et donc propager le signal à l'ensemble du système neuronal (le cerveau biologique). Un neurone ne va émettre une impulsion que si le signal transmis au corps cellulaire par les dendrites dépasse un certain seuil appelé "*seuil de déclenchement*". [29]

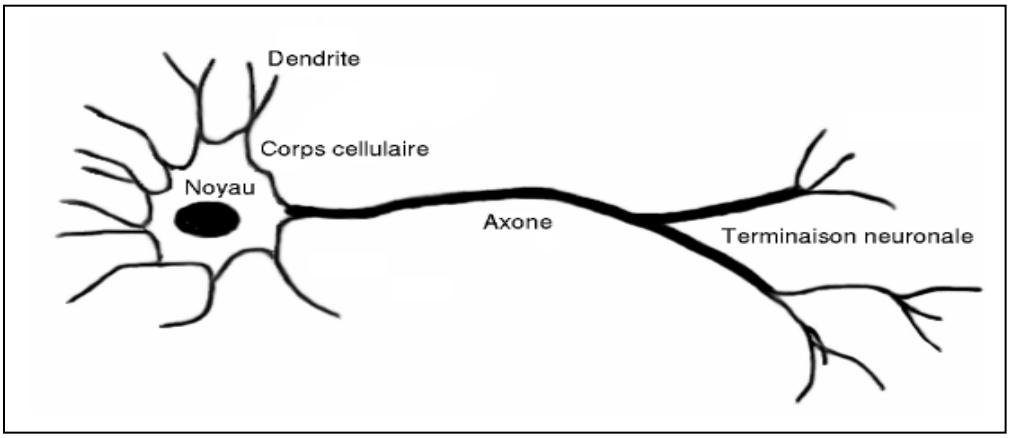


Figure 3.1: Neurone biologique.

4. Inspiration mathématique :

De la même façon que pour les algorithmes génétiques, les informaticiens se sont inspirés de la biologie pour créer des neurones formels afin de réaliser certaines opérations [3]. Un neurone formel est une entité dont le comportement peut généralement être représenté par une fonction mathématique et qui est relié à d'autres neurones formels en un réseau.

Le premier neurone formel est apparu en 1943. On le doit à *Mc Culloch* et *Pitts*. Voici un schéma de leur modèle de neurone formel : [32]

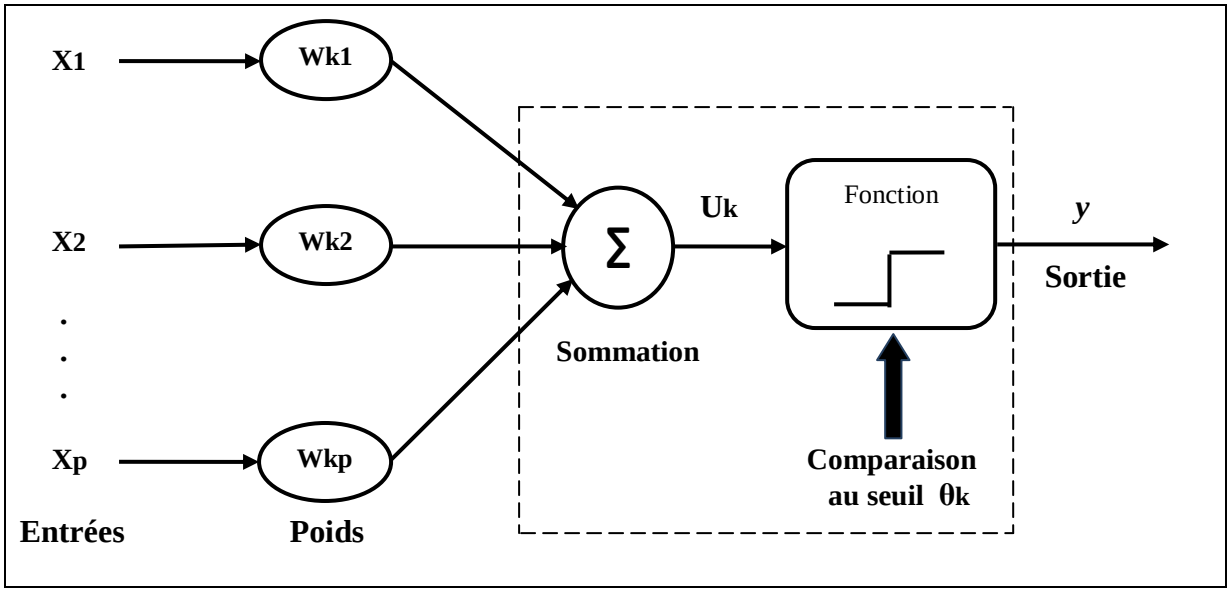


Figure 3.2: Structure d'un neurone formel Mc Culloch et Pitts.

Le neurone formel est donc une modélisation mathématique qui repend les principes du fonctionnement du neurone biologique, en particulier la *sommation* des entrées.

D'un point de vue mathématique, le neurone formel peut être représenté de la manière suivante : [33]

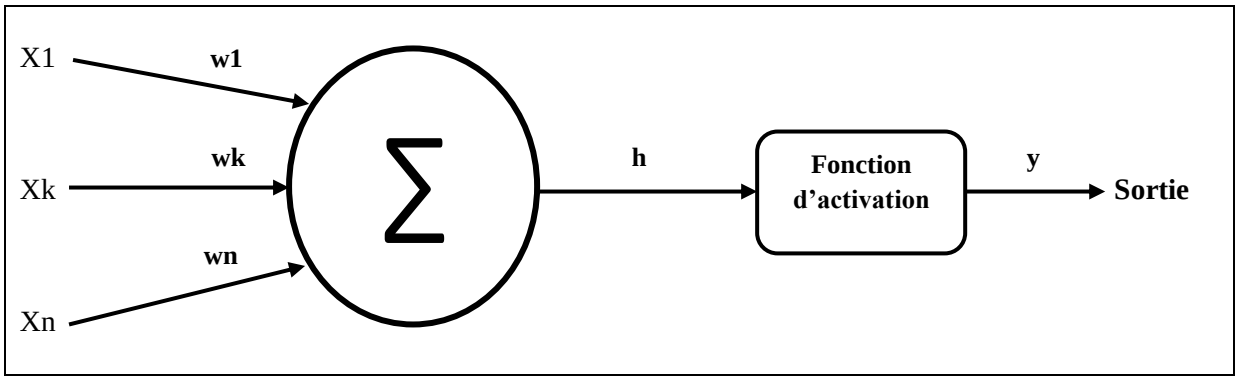


Figure 3.3: Neurone formel mathématique.

Le neurone formel possède généralement plusieurs entrées et une sortie qui correspondent respectivement aux dendrites et à l'axone du neurone. Il est coutumier d'attribuer des *poids* aux entrées afin de représenter l'importance des synapses et de pouvoir effectuer des

opérations plus complexes. Le neurone formel réalise ensuite le calcul de la valeur de la *sortie* à partir des valeurs des *entrées*. [32]

Pour un nombre compris entre 1 et un nombre quelconque *n*, le neurone formel va calculer la somme de ses entrée (*x1,…,xn*), pondérées par les poids synaptiques (*w1,…,w2*), d'où la formule, avec *f = fonction de transfert :*

$$Y = f\left(\sum_{j=1}^{n} wj\ xj\right)$$

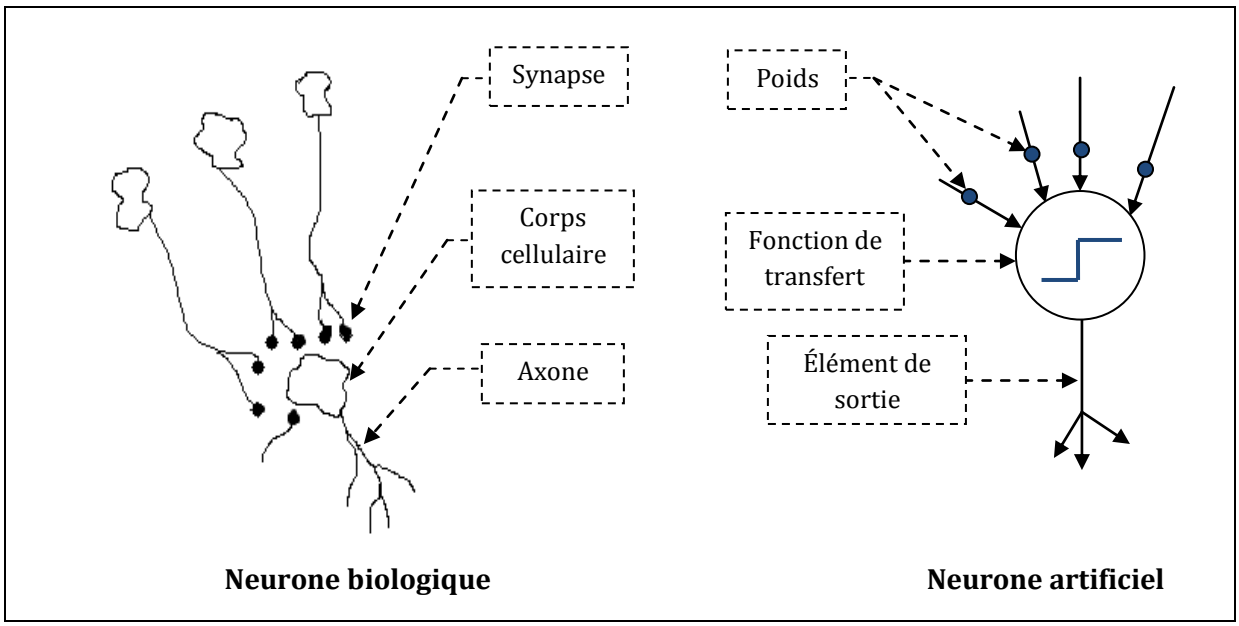


Figure 3.4 : Correspondance entre neurone biologique et neurone artificiel.

5. Réseaux de neurones artificiels :

Un réseau de neurone artificiel (RNA) est un ensemble de neurones formels associés en couches et fonctionnant en parallèle.

Dans un réseau, chaque couche fait un traitement indépendant des autres et transmet le résultat de son analyse à la couche suivante. L'information donnée au réseau va donc *se propager* couche par couche, de la couche d'entrée à la couche de sortie, en passant soit par aucune, une ou plusieurs couches intermédiaires "*couche cachées*".

Il est à noter qu'en fonction de l'algorithme d'apprentissage, il est aussi possible d'avoir une propagation de l'information à reculons (back propagation). Habituellement (excepté pour les couches d'entrée et de sortie), chaque neurone dans une couche est connecté à tous les neurones de la couche précédente et de la couche suivante.

Les RNA ont la capacité de stocker la connaissance empirique et de la rendre disponible à l'usage. Les habiletés de traitement (et donc de la connaissance) du réseau vont être stockées dans les poids synaptiques, obtenus par des processus d'adaptation ou d'apprentissage.

En ce sens, les RNA ressemblent donc au cerveau car non seulement, la connaissance est acquise au travers d'un apprentissage mais de plus, cette connaissance est stockée dans les connexions entre les entités, soit dans les poids synaptiques. [30]

6. Fonction d'activation :

Appelée aussi *Fonction de transfert*, le choix d'une fonction d'activation est un élément constitutif important des réseaux de neurones. Ainsi, l'identité n'est pas toujours suffisante, bien au contraire, et le plus souvent des fonctions non linéaires et plus évoluées seront nécessaires. [34]

À titre illustratif, le tableau suivant (tableau 3.1) présente quelques fonctions couramment utilisées comme fonctions d'activation, avec : $y = f(n)$.

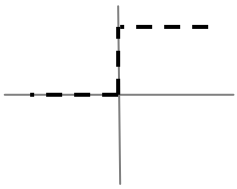
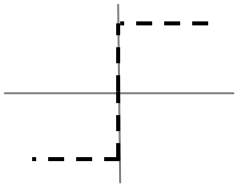
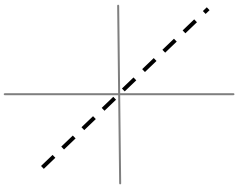
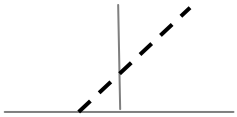
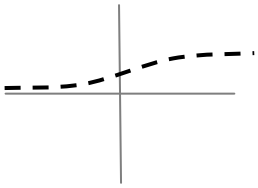
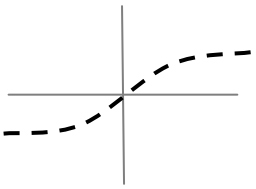
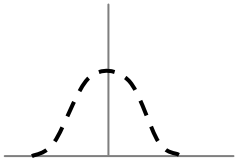
Nom de la fonction	Relation d'entrée/sortie	Courbe correspondante	Intervalle de définition
Seuil	$\begin{cases} y = 0, & n < 0 \\ y = 1, & n \geq 0 \end{cases}$		(0, 1)
Seuil symétrique	$\begin{cases} y = -1, & n < 0 \\ y = 1, & n \geq 0 \end{cases}$		(-1, 1)
Linéaire	y = n		)-∞, +∞(
Linéaire positive	$\begin{cases} y = 0, & n < 0 \\ y = n, & n \geq 0 \end{cases}$		(0, +∞(
Sigmoïde	y = 1 / (1 + Exp (-n))		(0, 1)
Tangente hyperbolique	y = 2 / (1 + Exp (-2n)) - 1		(-1, +1)
Gaussienne	y = Exp (- (n^2) / 2)		)0, +∞(

Tableau 3.1 : Fonctions de transfert.

7. Architecture des réseaux de neurones :

Architecture est le terme le plus général pour désigner la façon dont sont disposés et connectés les différents neurones qui composent un réseau. On peut classer les RNA en deux grandes catégories :

7.1. Les réseaux Feed-Forward :

Appelés aussi réseaux de type *perceptron*, ce sont des réseaux dans lesquels l'information se propage de couche en couche sans retour en arrière possible.

7.1.1. Les perceptrons :

7.1.1.1. Le perceptron monocouche :

C'est historiquement le premier RNA, proposé par Rosenblatt. C'est un réseau simple, puisque il ne se compose que d'une couche d'entrée et d'une couche de sortie.

Il a été conçu dans un but premier de reconnaissance des formes. Cependant, il peut aussi être utilisé pour faire de la classification et pour résoudre des opérations logiques simples telle "ET", "OU".

Sa principale limite est qu'il ne peut résoudre que des problèmes linéairement séparables. Il suit généralement un apprentissage *supervisé* selon la règle de *correction de l'erreur*.

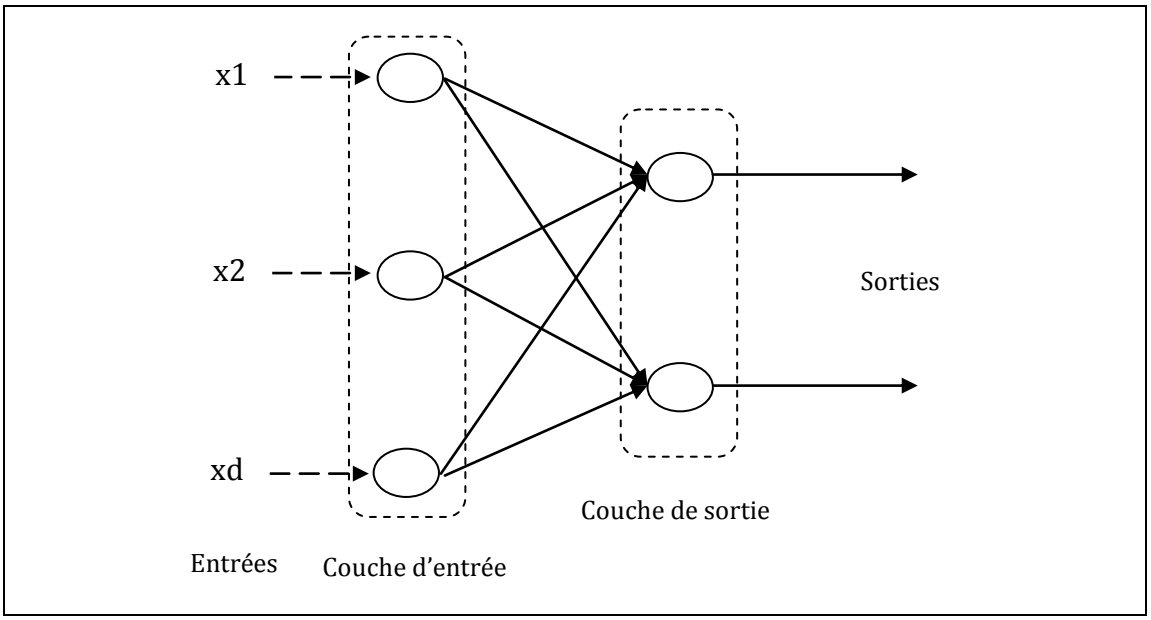


Figure 3.5: Schéma d'un perceptron mono-couche.

7.1.1.2. Perceptron multicouches :

C'est une extension du précédent, avec une ou plusieurs couches cachées entre l'entrée et la sortie. Chaque neurone dans une couche est connecté à tous les neurones de la couche précédente et la couche suivante (excepté pour la couche d'entrée et de sortie) et il n'y a pas de connexions entre les cellules d'une même couche. Les fonctions d'activation utilisé dans ce type de réseaux sont principalement les *fonctions à seuil* ou *sigmoïdes*.

Il peut résoudre des problèmes non linéairement séparables et des problèmes logiques plus compliqués. Il suit aussi un apprentissage *supervisé* selon la règle de *correction de l'erreur*.

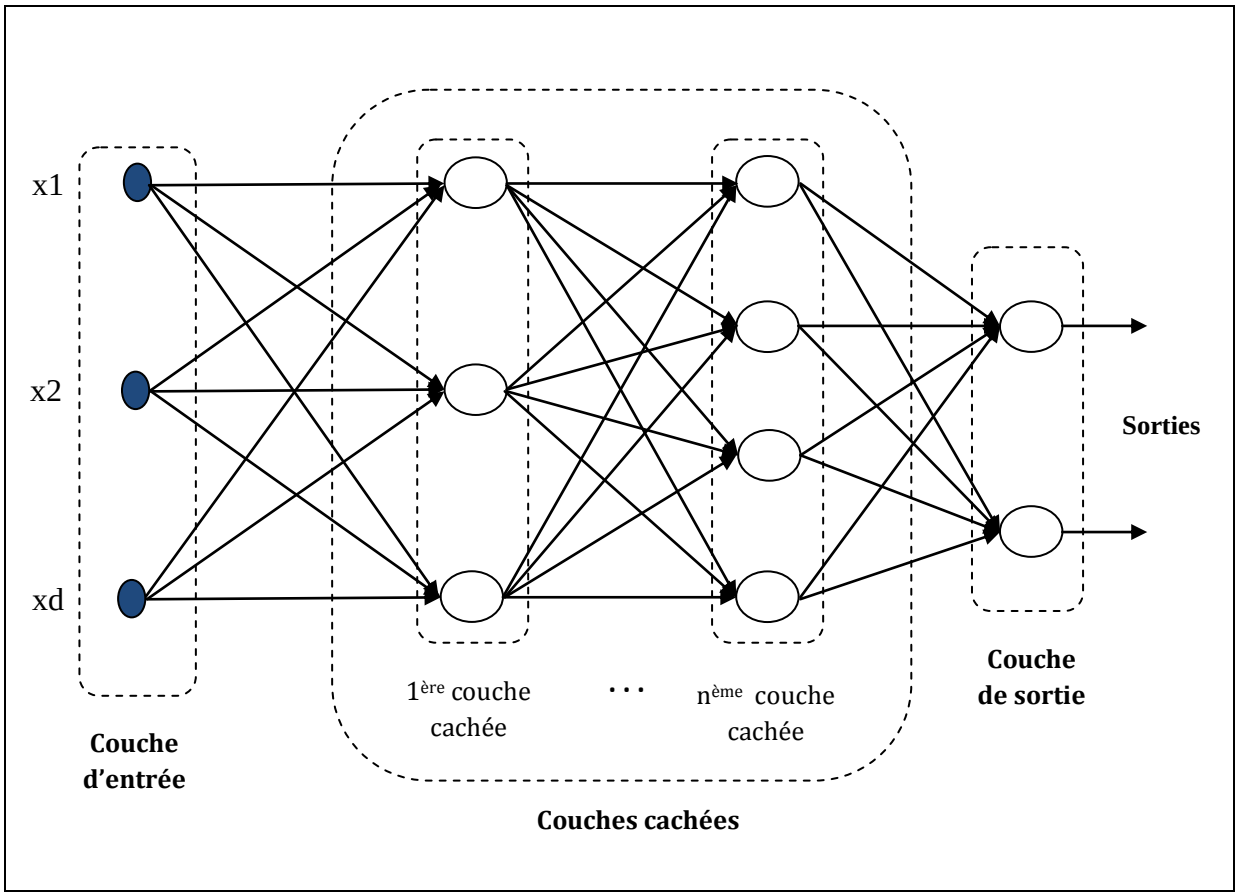


Figure 3.6: Schéma d'un perceptron multi couches.

7.1.2. Les réseaux à fonction radiale "RBF" :

L'architecture de ces réseaux est la même que pour les perceptrons multicouches (PMC) cependant, les fonctions de base utilisées ici sont des *fonctions gaussienne*. Les RBF seront donc employés dans les mêmes types de problèmes que les PMC à savoir, en classification et en approximation de fonctions, particulièrement.

L'apprentissage le plus utilisé pour les RBF est le mode *hybride* et les règles sont soit, la règle de *correction de l'erreur* soit, la règle *d'apprentissage par compétition*.

7.2. Les réseaux Feed-Back :

Appelés aussi *réseaux récurrents*, ce sont des réseaux dans lesquels il y a retour en arrière de l'information.

7.2.1. Les cartes auto-organisatrices de Kohonen :

Ce sont les réseaux à apprentissage *non supervisé* qui établissent une carte discrète, ordonnée topo logiquement, en fonction de patterns d'entrée. Le réseau forme ainsi une sorte de treillis dont chaque nœud est un neurone associé à un vecteur de poids. La correspondance entre chaque vecteur de poids est calculée pour chaque entrée. Par la suite, le vecteur de poids ayant la meilleure corrélation, ainsi que certains de ses voisins, vont être modifiés afin d'augmenter encore cette corrélation.

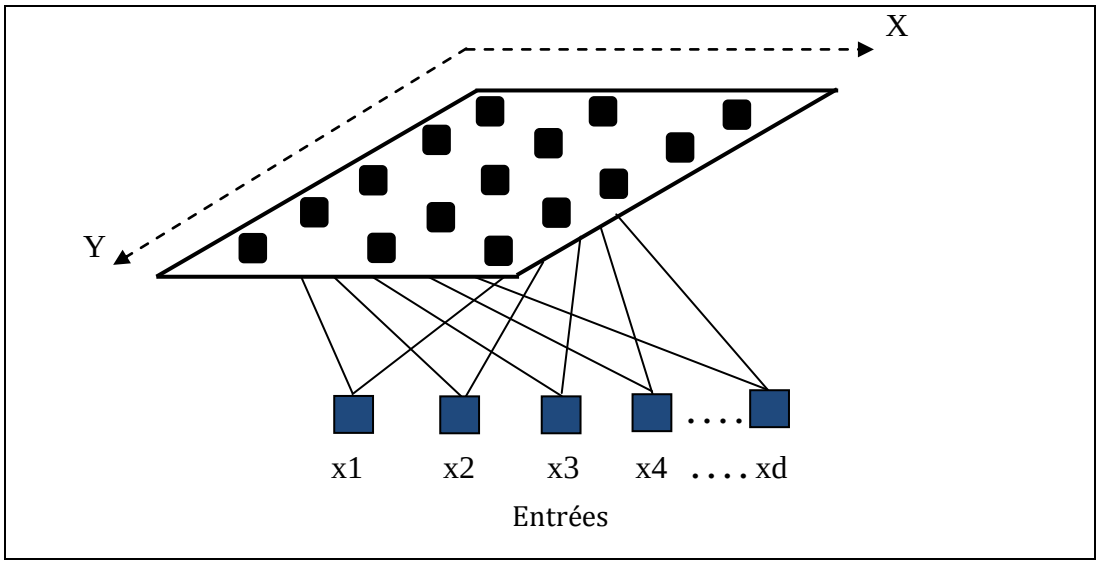


Figure 3.7: carte auto-organisatrice à deux dimensions.

7.2.2. Les réseaux de Hopfield :

Les Hopfield sont des réseaux récurrents et entièrement connectés. Dans ce type de réseau, chaque neurone est connecté à chaque autre neurone et il n'y a aucune différenciation entre les neurones d'entrée et de sortie. Ils fonctionnent comme une mémoire associative non-linéaire et sont capables de trouver un objet stocké en fonction de représentations partielles ou bruitées.

L'application principale des réseaux de Hopfield est l'entrepôt de connaissances mais aussi la résolution de problèmes d'optimisation. Le mode d'apprentissage utilisé ici est le mode *non supervisé.*

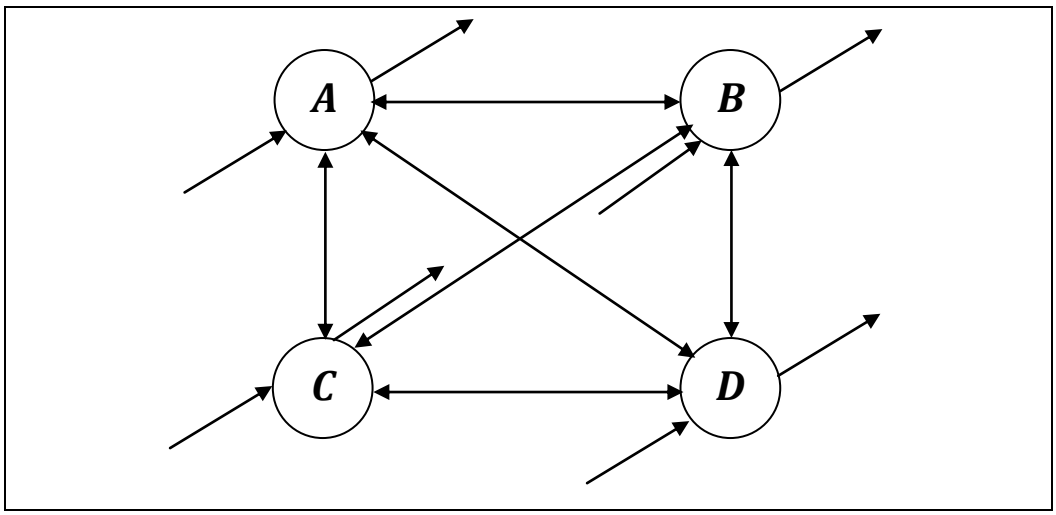


Figure 3.8 : Réseau de Hopfield à 4 neurones.

7.2.3. Les réseaux ART :

Les réseaux ART "*Adaptative Resonnance Theory*" sont des réseaux à apprentissage par compétition. Le problème majeur qui se pose dans ce type de réseaux est le dilemme "stabilité/plasticité". En effet, dans un apprentissage par compétition, rien ne garantit que les catégories formées restent stables. La seule possibilité, pour assurer la stabilité, serait que le coefficient d'apprentissage tende vers le zéro, mais le réseau perdrait alors sa plasticité.

Les ART ont été conçus spécifiquement pour contourner ce problème. Dans ce genre de réseau, les vecteurs de poids ne seront adaptés que si l'entrée fournie est suffisamment

proche, d'un prototype déjà connu par le réseau. On parlera alors de résonnance. A l'inverse, si l'entrée s'éloigne trop des prototypes existants, une nouvelle catégorie va se créer, avec prototype, l'entrée qui a engendrée sa création.

Il est à noter qu'il existe deux principaux types de réseaux ART : les ART-1 pour des entrées binaires et les ART-2 pour des entrées continues. Le mode d'apprentissage des ART peut être *supervisé ou non*.

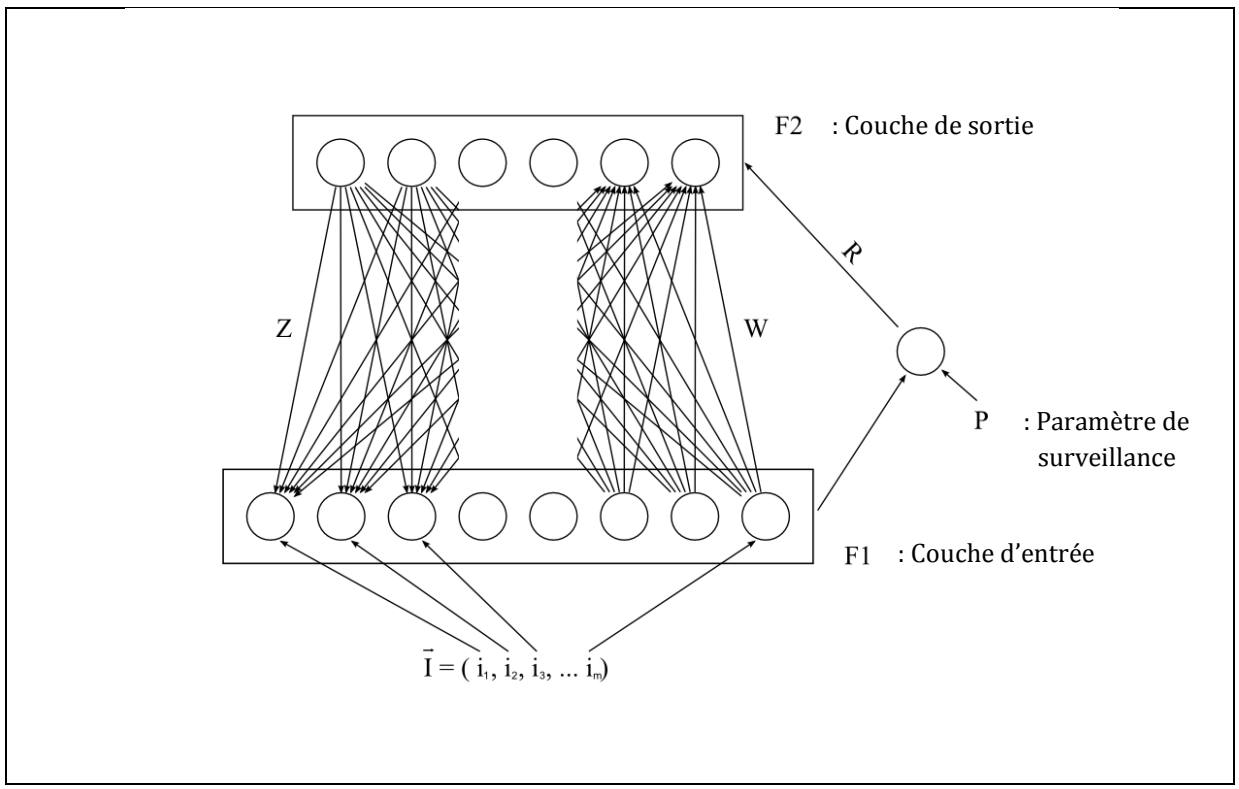


Figure 3.9: Réseau ART.

8. Domaines d'application des réseaux de neurones :

Le tableau suivant (tableau 3.2) résume, en général, la correspondance entre le type de réseaux et les applications les plus appropriées à ce dernier. [35]

Caractéristiques fonctionnelles	*Type des réseaux de neurones*
Reconnaissance de formes	MLP, Hopefield, Kohonen
Mémoires associatives	MLP, Hopefield, Kohonen
Optimisation	Hopefield, ART
Approximation de fonction	MLP, RBF
Modélisation et Control	MLP
Traitement d'image	Hopefield
Classification et clustrering	MLP, ART, RBF, Kohonen

Tableau 3.2 : Correspondance Réseaux de neurones – Domaines d'application

9. L'Apprentissage dans les réseaux de neurones :

Pour un RNA, l'apprentissage peut être considéré comme le problème de la mise à jour des poids des connexions au sein du réseau, afin de réussir la tâche qui lui est demandée. L'apprentissage est la caractéristique principale des RNA et il peut se faire de différentes manières et selon différente règles. [30]

9.1. Types d'apprentissage :

9.1.1. Le mode supervisé :

Dans ce type d'apprentissage, le réseau s'adapte par comparaison entre le résultat qu'il a calculé, en fonction des entrées fournies, et la réponse attendue en sortie. Ainsi, le réseau va se modifier jusqu'à ce qu'il trouve la bonne sortie, c'est-à-dire celle attendue, correspondant à une entrée donnée.

9.1.2. Le renforcement :

Le renforcement est en fait une sorte d'apprentissage supervisé et certains auteurs le classe d'ailleurs, dans la catégorie des modes supervisés. Dans cette approche le réseau doit

apprendre la corrélation entrée/sortie via une estimation de son erreur, c'est-à-dire du rapport échec/succès. Le réseau va donc tendre à maximiser un index de performance qui lui est fourni, appelé *signal de renforcement*. Le système étant capable ici, de savoir si la réponse qu'il fournit est correcte ou non, mais il ne connait pas la bonne réponse.

9.1.3. Le mode non-supervisé :

Appelé aussi auto-organisationnel, dans ce cas, l'apprentissage est basé sur des probabilités. Le réseau va se modifier en fonction des régularités statistiques de l'entrée et établir des catégories, en attribuant et en optimisant une valeur de qualité, aux catégories reconnues.

9.1.4. Le mode hybride :

Le mode hybride reprend en fait les deux autres approche, puisque une partie des poids va être déterminée par apprentissage supervisé et d'autre partie par apprentissage non-supervisé.

9.2. Règles d'apprentissage :

9.2.1. Règle de correction d'erreurs :

Cette règle s'inscrit dans le paradigme d'apprentissage supervisé, c'est-à-dire dans le cas où l'on fournit au réseau une entrée et la sortie correspondante.

Si on considère "*y*" comme étant la sortie calculée par le réseau, et "*d*" la sortie désirée, le principe de cette règle est d'utiliser l'erreur (*d-y*), afin de modifier les connexions et de diminuer ainsi l'erreur globale du système. Le réseau va donc s'adapter jusqu'à ce que "*y*" soit très proche de "*d*". Ce principe est notamment utilisé dans le modèle du perceptron simple.

9.2.2. Apprentissage de Boltzmann :

Les réseaux de Boltzmann sont des réseaux symétriques récurrents. Ils possèdent deux sous-groupes de cellules, le premier étant relié à l'environnement "*cellules visibles*" et le second ne l'étant pas "*cellules cachées*". Cette règle d'apprentissage est de type stochastique (qui relève partiellement du hasard) et elle consiste à ajuster les poids des connexions, de telle sorte que l'état des cellules visibles satisfasse une distribution probabiliste souhaitée.

9.2.3. Règle de Hebb :

Cette règle est basée sur des données biologiques, modélise le fait que si des neurones, de part et d'autre d'une synapse, sont activés de façon synchrone et répétée, la force de la connexion synaptique va aller croissant. Il est à noté que l'apprentissage est localisé, c'est-à-dire que la modification d'un poids synaptique "*Wij*" ne dépend que de l'activation d'un neurone "*i*" et d'un autre neurone "*j*".

9.2.4. Règle d'apprentissage par compétitions :

La particularité de cette règle, c'est qu'ici l'apprentissage ne concerne qu'un seul neurone. Le principe de cet apprentissage est de regrouper les données en catégories. Les patrons similaires vont donc être rangés dans une même classe, en basant sur les corrélations des données, et seront représentés par un seul neurone, on parle de "*winner-take-all*".

Dans un réseau à compétition simple, chaque neurone de sortie est connecté aux neurones de la couche d'entrée, aux autres cellules de la couche de sortie (connexions inhibitrices) et à elle-même (connexion excitatrice). La sortie va donc dépendre de la compétition entre les connexions inhibitrices et excitatrices. **[**30]

10. Les méthodes d'apprentissage :

Dans les systèmes experts, les connaissances de l'expert ont une forme énumérée : elles sont exprimées sous forme de règles. Dans le cas des réseaux de neurones, les connaissances ont une forme distribuée : elles sont codées dans les poids des connexions, la topologie du réseau, les fonctions de transfert de chaque neurone, le seuil de ces fonctions, la méthode d'apprentissage utilisée. [36]

Il existe un certain nombre de méthodes d'apprentissage, et dans ce travail nous nous intéressons à la méthode de *rétro-propagation du gradient de l'erreur*.

10.1. La rétro-propagation du gradient de l'erreur :

10.1.1. Représentation :

Cet algorithme est utilisé dans les réseaux de type Feed-Forward, c'est l'outil le plus utilisé actuellement dans le domaine des réseaux de neurones. Le principe de la rétro-propagation consiste à présenter au réseau un vecteur d'entrées, de procéder au calcul de la sortie par

propagation à travers les couches, de la couche d'entrée vers la couche de sortie en passant par les couches cachées. Cette sortie obtenue est comparée à la sortie désirée, une *erreur* est alors obtenue.

A partir de cette erreur, est calculé le *gradient de l'erreur* qui est à son tour propagé de la couche de sortie vers la couche d'entrée, d'où le terme de rétro-propagation. Cela permet la modification des poids du réseau et donc l'*apprentissage*. L'opération est réitérée pour chaque vecteur d'entrée et cela jusqu'à ce que le critère d'arrêt soit vérifié. [36]

L'algorithme de rétro-propagation du gradient de l'erreur est basé sur la *minimisation de l'erreur quadratique* "E" calculée en fonction des *n* sorties désirées $\mathbf{yd_i}$ et des *n* sorties effectivement données **yi** par le réseau [37]:

$$\mathbf{E_i = \sum_{i=1}^{n} (yi - ydi)^2}$$

Minimiser cette énergie revient alors à modifier les poids des connexions de la manière suivante :

$$\Delta W_{kh}^{(j)} = - a. \delta_{k}^{(j)} . y_{h}^{(j-1)}$$

Avec *a* le gain d'adaptation, $\delta_{K}^{(j)}$ l'erreur du neurone *k* de la couche *j* et $y_{h}^{(j-1)}$ la sortie du neurone *h* de la couche *j-1*.

Pour les neurones de la dernière couche :

$$\delta_{K}^{(j)} = y_{k}^{(j)} - yd_{k}$$

Pour les neurones des couches internes :

$$\delta_{k}^{(j)} = [\sum_{i \in couche\ (j+1)} \delta_{i}^{(j+1)} . w_{ik}^{(j+1)}]. \sigma' p_{k}^{(j)}$$

Avec :

$$p_k^{(j)} = \sum_i w_{ki} . x_i$$

Où x_i correspond à la sortie du neurone *i* et :

$$\sigma p_k^{(j)} = \frac{1}{1 + exp(-(p_k^{(j)}))}$$

Et σ' sa dérivée.

10.1.2. Initialisation des poids :

L'initialisation des poids avant l'application de l'algorithme d'apprentissage par rétro-propagation du gradient est importante. Cette initialisation influe sur la vitesse de convergence du réseau mais aussi sur la qualité du réseau obtenu**.** [38]

10.1.3. Temps d'apprentissage :

Le temps de convergence d'un réseau dépend de l'espace initial de représentation des poids et de l'espace final après convergence. En effet, plus les poids initiaux sont proches de leur valeur finale plus la convergence est rapide. On peut distinguer deux types de méthodes dans la littérature:

- Les méthodes d'initialisation aléatoire dans un intervalle choisi de manière adéquate.
- Les méthodes basées sur des techniques non aléatoires. [39]

Plusieurs recherches ont été effectuées dans le même domaine, on distingue quelques uns dans ce qui suit :

- *Burel G* et *Falhman S.E,* proposent de choisir des poids de manière uniforme dans un intervalle dépendant des données à apprendre. *Fahlman* propose des intervalles variant de [-4.0, 4.0] à [-0.5, 0.5] selon les données d'apprentissage. [40]

- *Bottou L-Y,* propose d'initialiser les poids dans un intervalle [–a√ din,a√ din] , où *"a"* est calculé de sorte que la variance des poids corresponde au point où la pente de la tangente de la fonction d'activation est maximum, et *"din"* le nombre d'unités de la couche précédente. [41]
- *Lee* et *al*, ont montré théoriquement que la saturation prématurée des neurones (variations des poids trop petites pour avoir un effet sur la sortie des neurones) augmentait avec les valeurs maximales des poids. Ils en concluent que des valeurs initiales petites accroissent la vitesse de convergence mais aussi que des valeurs trop petites dégradent cette vitesse. [42]
- D'autres travaux sont basés sur des schémas d'initialisation et de méthode pseudo-inverse, où seule la première couche est initialisée aléatoirement, les autres couches étant initialisées en fonction des sorties produites par la couche qui les précède. *Denoeux* et *Lengellé*, proposent une méthode basée sur des données prototypes permettant d'accélérer la vitesse de convergence des réseaux. [4]

10.1.4. Qualité du réseau résultant :

L'apprentissage par rétro-propagation du gradient peut être vu comme l'optimisation d'une fonction ayant les poids du réseau pour paramètres. Ceci mène à une convergence de la fonction vers un minimum local qui peut être aussi global. Lorsque c'est le cas, on peut considérer que l'apprentissage a été fait correctement. Lorsque ce n'est pas le cas, le réseau obtenu n'est pas optimal. Si dans la plupart des cas, la solution trouvée est très proche de la solution optimale et donc acceptable, il reste des cas où la solution est médiocre.

Des études ont été faites pour initialiser les poids de manière adéquate aux données à apprendre [43]. Denoeux et Lengellé, montrent des résultats plus robustes lorsque l'apprentissage est effectué avec des prototypes [44].

10.1.5. Limitations de la méthode de rétro-propagation :

Bien que l'algorithme de rétro-propagation soit le plus utilisé pour l'apprentissage supervisé des perceptrons multicouches, son implantation se heurte à plusieurs difficultés techniques [45]. Il n'existe pas de méthodes permettant de :

- Trouver une architecture appropriée (nombres de couches, nombre de neurones).
- Choisir une taille et une qualité adéquate d'exemples d'entraînement.
- Choisir des valeurs initiales satisfaisantes pour les poids, et des valeurs convenables pour les paramètres d'apprentissage permettant d'accélérer la vitesse de convergence.

- Problème de la convergence vers un minimum local, qui empêche la convergence et cause l'oscillation de l'erreur.
- La performance de cette approche diminue rapidement en fonction de la taille (complexité).
- Basée sur une recherche de gradient, cette approche est inapplicable si des connexions sont manquantes (fonctions discontinues).

11. Conclusion :

Les réseaux de neurones (*RNA*) constituent une manière simplifiée de simuler les capacités des organismes vivants à s'adapter à leur environnement par apprentissage. De nombreux chercheurs ont étudié le couplage des réseaux de neurones et des algorithmes évolutionnaires, Un grand nombre de travaux ont été effectués sur cette évolution. Semble-t-il le couplage de ces deux métaphores biologiques donnent des résultats de loin meilleurs que de les utiliser indépendamment.

Dans le chapitre suivant, on va aborder la manière de combinaison de ces deux méthodes afin de réaliser un environnement de développement des stratégies de jeux.

CHAPITRE IV

ANALYSE ET CONCEPTION

Pour mener à bien le projet, nous devons tout naturellement avoir recours à un formalisme de conception. Cette partie est consacrée aux étapes fondamentales pour le développement de notre système.

1. Introduction :

À partir de tous ce que nous avons déjà vu dans les chapitres précédents, nous pouvons maintenant faire une *hybridation* (combinaison) du calcul évolutionnaire et le calcul neuronal afin d'aboutir à une méthode efficace et optimale pour développer et améliorer les stratégies des jeux combinatoire. Pour appliquer cette approche, on a choisi le jeu de dames.

Ce travail est inspiré des travaux des deux chercheurs américains *David B.Fogel et Kumar Chellapilla*. Pour cela, nous avons appuyé sur le document qui décrit ce projet [5], il présente une nouvelle approche pour développer les stratégies des jeux sans avoir besoin de connaissances préalables, il représente également le résultat de la théorie évolutionnaire à savoir la sélection naturelle en appliquant le principe du « meilleur qui survie ».

Pour la réalisation de notre logiciel, on a suivi la démarche illustrée dans le schéma suivant :

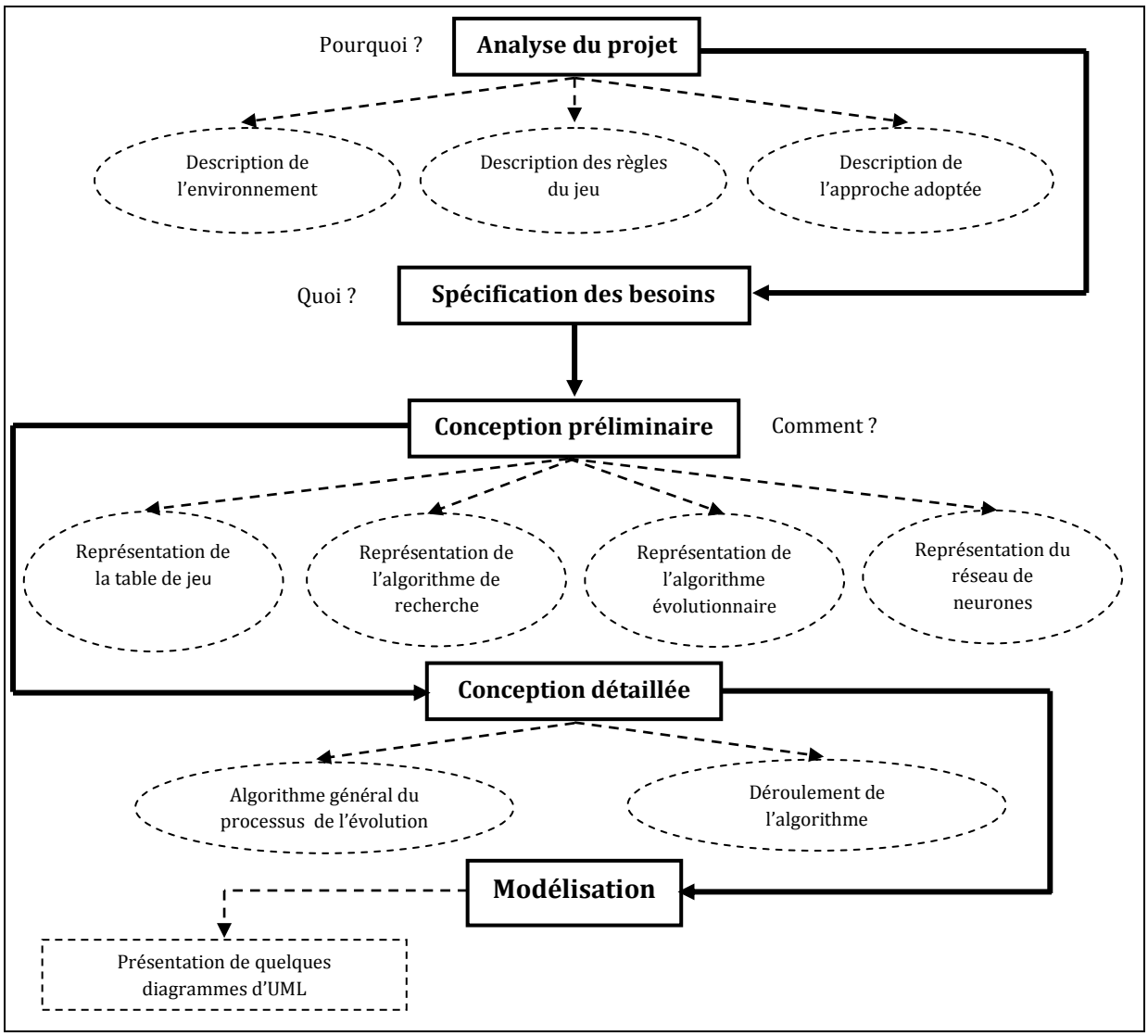


Figure 4.1 : Schéma générale du chapitre

2. Analyse du projet :

2.1. Description de l'environnement :

Le *système* que l'on va concevoir est un environnement de création et d'amélioration des stratégies du jeu de dames. Il pourra donc *interagir* avec un *administrateur* pour configurer et créer les stratégies employées par les différents adversaires machines ou bien tout simplement avec un *joueur* pour jouer une partie de jeu de dames contre la machine, ce qui constitue l'*environnement* lié au système.

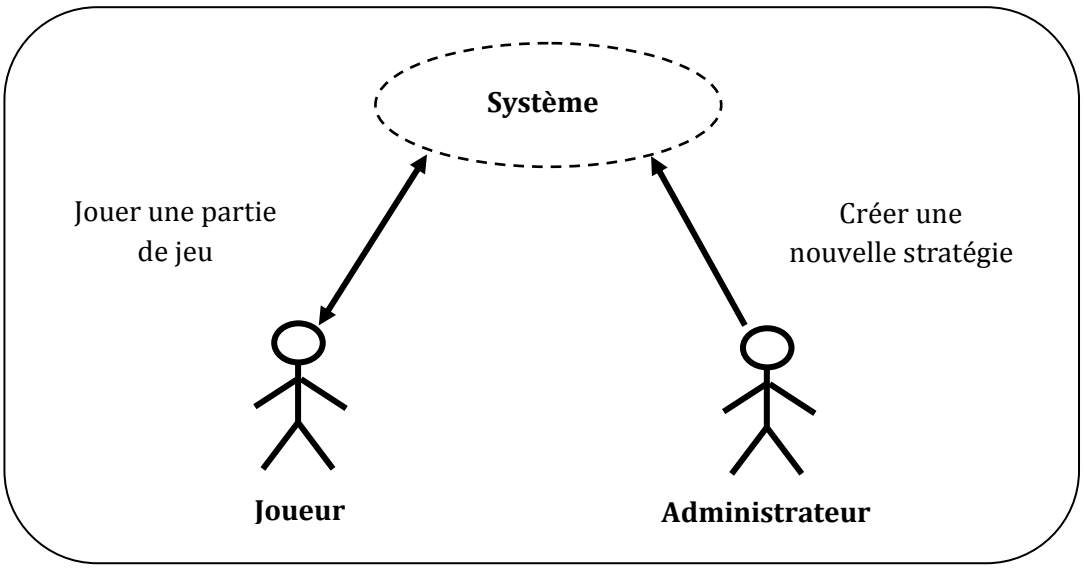


Figure 4.2 : Schéma de l'environnement.

2.2. Description des règles du jeu :

2.2.1. But du jeu :

Capturer ou immobiliser les pièces de son adversaire.

2.2.2. Matériel :

Le jeu de dame est traditionnellement joué sur une table huit par huit (8x8) avec des cases de couleurs alternatives.

Il y a deux joueurs, dénotés par les pièces noires et blanches respectivement. Chaque coté (joueur) a 12 pions qui sont placés dans les 12 premières cases alternatives de la même couleur, avec la case extrême droite sur la première rangée de chaque joueur (coté) étant

laissée ouverte, c'est-à-dire cette case appartient à l'ensemble de cases où on ne peut pas y mettre les pièces, comme le montre la figure suivante :

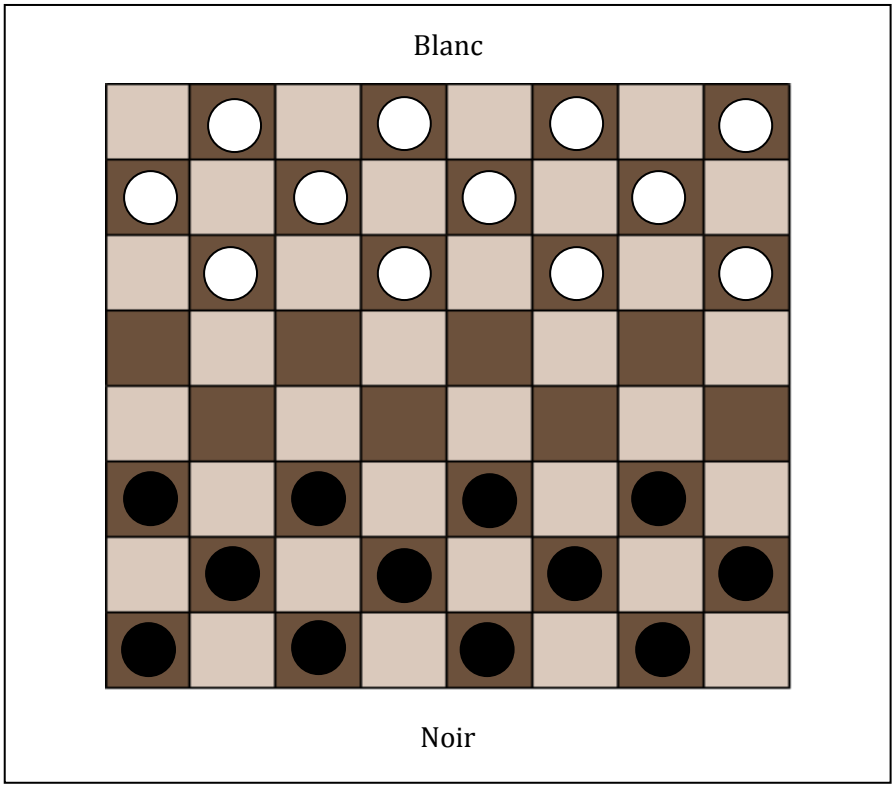


Figure 4.3 : Position initiale du jeu de dames.

2.2.3. Comment joué :

2.2.3.1. Le déplacement :

Le joueur noir se déplace d'abord, ensuite le jeu s'alterne entre les cotés. On peut faire avancer diagonalement les pièces une case à la fois.

2.2.3.2. L'enlèvement :

Lorsqu'une pièce se positionne à coté d'une pièce d'opposition (adversaire), et il y a une case vide derrière cette dernière, il faut sauter diagonalement au dessus de la pièce d'opposition. Dans ce dernier cas, le pion d'opposition est pris et supprimé du jeu. Une prise peut s'effectuer vers l'avant ou vers l'arrière.

Si un saut placerait à leur tour la pièce sautant en une position pour un autre saut, ce saut doit également être joué, et ainsi de suite jusqu'aucun saut ne soit disponible pour cette pièce.

Toute les fois qu'un saut est disponible, il doit être joué de préférence à un mouvement qui ne saute pas, cependant quand des mouvements multiples sont disponibles, le joueur a le choix d'un saut à faire. Le mieux choix est le saut qui offre l'enlèvement de plusieurs pièces de l'adversaire (c'est-à-dire un double saut contre un saut simple).

Quand une pièce avance à la dernière rangée de la table, elle devient un **Roi**, et peut ensuite se déplacer diagonalement à n'importe quelle direction (c'est-à-dire en avant ou vers l'arrière).

2.2.4. Résultat du jeu :

Le jeu se termine quand un joueur n'a plus de mouvements disponibles, qui se produit le plus souvent en faisant enlever leur dernière pièce de la table. Mais il peut également se produire quand toutes les pièces existantes sont emprisonnées. Ceci implique la perte pour ce joueur sans mouvement restant et une victoire pour l'adversaire (l'objectif du jeu). Le jeu peut également se terminer lorsque la partie est nulle.

2.3. Description de l'approche adoptée :

Ce travail relève de la théorie des jeux, les algorithmes évolutionnaires et les réseaux de neurones. Le but est de réaliser un environnement pour le développement et l'amélioration des stratégies de jeux, doté d'une intelligence incrémentale, en se basant sur l'hybridation des réseaux de neurones avec des algorithmes évolutionnaire, tout en essayant d'atteindre un idéal tant dans les résultats que dans la conception du logiciel.

L'intérêt de l'hybridation des réseaux de neurones avec des algorithmes évolutionnaires se base sur l'observation qu'une recherche locale par une méthode de descente de gradient (réseaux de neurones) est bien complémentée par une recherche globale effectuée par des algorithmes évolutionnaires. [1]

3. Spécification des besoins :

C'est une étape primordiale au début de chaque démarche de développement. Son but est de veiller à développer un logiciel adéquat, sa finalité est la description générale des fonctionnalités du système, en répondant à la question : Quelles sont les fonctions du système ?

Les fonctions principales que le logiciel devra satisfaire sont les suivantes :

- Permettre à un utilisateur de créer une nouvelle stratégie de jeu en passant par les étapes qu'on va discuter dans la suite de ce chapitre.
- Permettre à un utilisateur de jouer une partie de jeu de dames contre l'ordinateur, c'est-à-dire contre la meilleure stratégie sélectionné expérimentalement.

Nous avons identifié les problèmes suivants :

- L'explosion combinatoire qui reste un défi pour les gens intéressés de la théorie des jeux.
- La difficulté de déterminer une fonction d'évaluation d'une situation possible dans le jeu.
- La difficulté de trouver une meilleure représentation des stratégies du jeu.

4. Conception préliminaire :

La conception préliminaire permet de déterminer l'architecture informatique globale du système.

4.1. Représentation de la table du jeu :

Pour la réalisation de ce travail, chaque table de jeu a été représentée par un vecteur de longueur 32 dont chaque composant correspond à une position disponible dans la table. Les composants dans le vecteur pourraient prendre des valeurs appartenant à l'ensemble **{-k, -1, 0, +1, +k}** Où :

- "k" est la valeur assignée pour le Roi.
- "1" est la valeur assignée pour la pièce.
- "0" est la valeur assignée pour une case vide.
- Le signe de la valeur (+/-) indique si la pièce en question appartient au joueur (positif) ou à l'adversaire (négatif).

***Exemple* :** Au début du jeu, le vecteur initial sera comme suit :

(-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0 , 0, 0, 0, 0, 0, 0, 0, +1, +1, +1, +1, +1, +1, +1, +1, +1, +1, +1, +1)

Ainsi, chaque table a été découpée en sous sections (nxn) dans le but de fournir des informations spatiales d'adjacence ou de proximité, c'est-à-dire si deux cases sont des voisines, prés l'une l'autre, ou distantes.

On peut donc construire :

- **36** sous sections de (3x3).
- **25** sous sections de (4x4).
- **16** sous sections de (5x5).
- **09** sous sections de (6x6).
- **04** sous sections de (7x7).
- **01** seule section de (8x8).

Donc il ya en totale 91 sous sections possibles qu'on peut construire à partir de la table. La figure suivante montre des échantillons de (3x3), (4x4) et (5x5) sous sections carrées :

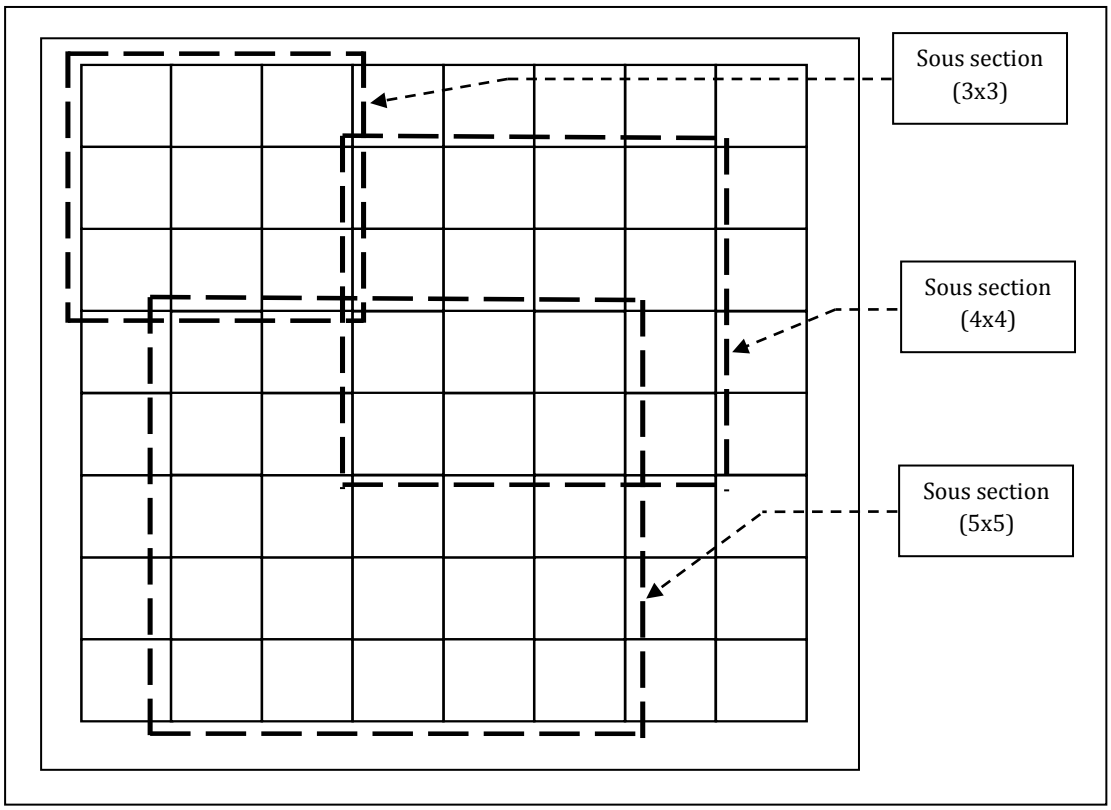


Figure 4.4 : Découpage de la table d'un jeu de dame en sous sections.

4.2. Représentation du réseau de neurones :

4.2.1. Architecture du réseau de neurones :

Trouver l'architecture adéquate d'un réseau de neurones à un problème donné n'est pas une chose facile. Trouver le nombre optimal de couches cachées ainsi que le nombre de neurones dans chaque couche et leurs connexions se fait plus de manière empirique que par une méthode basée sur un fondement théorique. [4]

Les réseaux de neurones auxquels nous nous intéressons sont des *perceptrons multicouches* avec *apprentissage par rétro-propagation du gradient*. En l'occurrence, nous nous sommes attachés à étudier des réseaux de neurones composés des couches suivantes :

- Une couche d'entrée.
- Trois couches cachées.
- Une couche de sortie contenant un seul neurone.

La deuxième, la troisième couche cachée et la couche de sortie ont une structure *entièrement reliée,* c'est-à-dire tous les neurones d'une couche sont reliés à tous les neurones de la couche suivante, tandis que les connexions de la première couche cachée étaient spécialement conçues pour capturer l'information spatiale de la table du jeu.

La *fonction de transfert* de chaque nœud cachée et de sortie était la *tangente hyperbolique*, elle a été choisie pour les raisons suivantes :

- C'est une fonction dérivable vers l'infini.
- C'est une fonction continue
- C'est une fonction non linéaire.

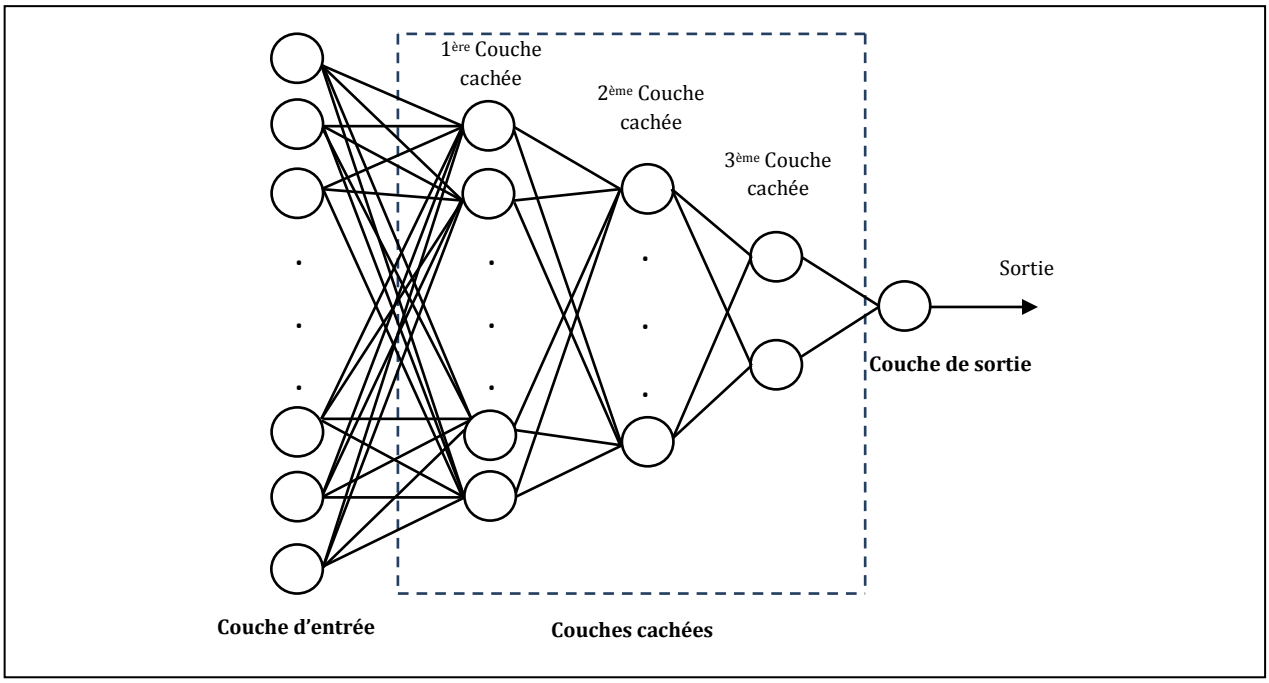


Figure 4.5 : Architecture du Réseau de neurones.

4.2.2. Les applications des réseaux de neurones :

4.2.2.1. Générer les caractéristiques spatiales de la table :

Les efforts précédents ont utilisé pour traiter les entrées de la table un réseau de neurones avec deux couches cachées comportant 40 et 10 nœud respectivement. Ainsi, la table (8x8) a été interprétée simplement en tant que vecteur (1x32), et le réseau de neurones a été forcé pour apprendre toutes les caractéristiques spatiales de la table. [5]

Pour ne pas handicaper la procédure d'apprentissage de cette manière, le réseau de neurones utilisé ici a mis en application une série de 91 nœuds de prétraitement qui recouvrent (nxn) sous sections de la table.

Ainsi, pour permettre au réseau de neurones de générer les caractéristiques spatiales de la table entière du jeu depuis les sous sections qui pourrait par conséquent être traité dans les couches cachées suivante, les 36 sous sections possibles de **(3x3)** ont été fournies comme entrée aux 36 premiers nœuds (neurones) dans la première couche cachée.

Les 25 sous sections possibles de **(4x4)** ont été assignées aux 25 prochains nœuds dans cette couche, et ainsi de suite. En fin de compte, on obtiendra 91 sous sections qui ont été données comme entrée aux 91 nœuds de la première couche cachée.

Le schéma suivant montre la structure du réseau de neurones :

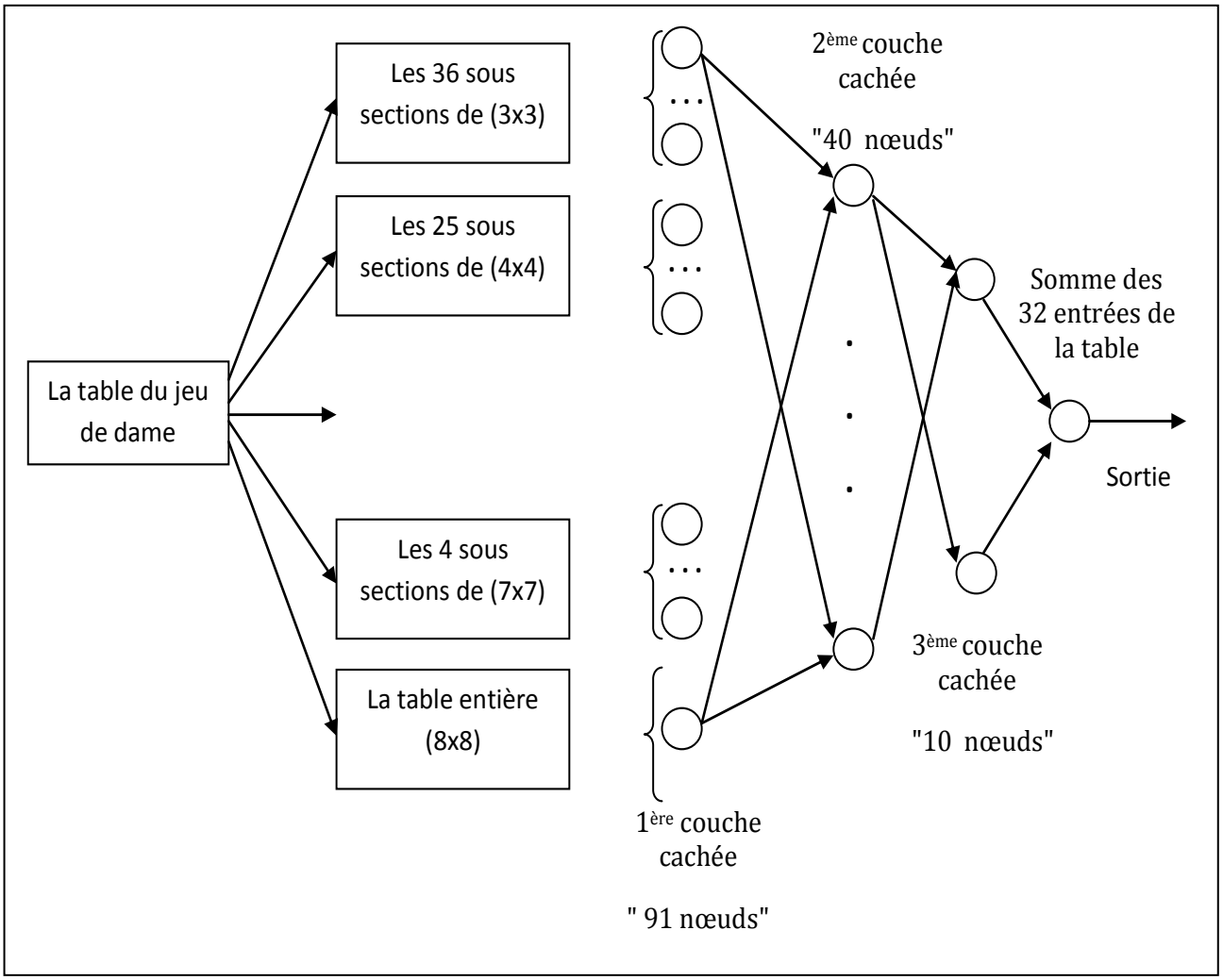


Figure 4.6 : Structure du réseau de neurones

4.2.2.2. Évaluation du jeu :

Comme nous avons déjà vu dans le premier chapitre « *Introduction à la théorie des jeux* », le mouvement d'un joueur est déterminé en évaluant la qualité présumée des futures positions potentielles (présentées dans l'arbre du jeu). Pour cela, et afin d'évaluer notre jeu, on présente le vecteur qui représente les positions des pièces dans la table du jeu au réseau de neurones pour l'évaluation.

Lorsque la table du jeu est présentée au réseau de neurones pour l'évaluation, sa sortie scalaire a été interprétée comme valeur (représentante) de cette table depuis la position du joueur dont les pièces ont été dénotées par des valeurs positives. Plus la sortie est proche de

+1, plus l'évaluation de l'entrée (la table du jeu) est *meilleure*. De même, plus la sortie est proche de **-1**, plus la table est *mauvaise.* Toutes les positions qui étaient des victoires pour le joueur (par exemple, aucune pièces d'oppositions restantes) ont été assignées la valeur exactement de **+1** et de même toutes les positions qui étaient des pertes ont été assignées la valeur **-1**.

Remarque **:** Il est important de noter immédiatement que nous n'avons pas tenté d'offrir au réseau de neurones des caractéristiques utiles comme entrées.

4.3. Représentation de l'Algorithme évolutionnaire :

Dans le chapitre 02 «*Les algorithmes évolutionnaires : principes et méthodes* », nous avons présenté deux méthodes classiques des EA, on a étudié leurs caractéristiques afin de choisir une méthode pour la réalisation de notre projet, notre choix c'est tombé sur les *Stratégies d'évolution*.

La représentation d'un réseau connexionniste dans un génotype a suscité un certain nombre de travaux. A un extrême, le réseau est codé littéralement dans le génotype, dont chaque poids est codé d'une manière précise. Dans ce cas, les Stratégies d'évolution se réduisent à un problème d'optimisation multicritères standard.

4.4. Représentation de l'algorithme de recherche :

Le principe des algorithmes de recherche dans les jeux est d'évaluer les différents coups à jouer pour un joueur donné, et de retourner le meilleur. Ceci implique de connaître l'ensemble des coups jouables par le joueur on construisant ce qu'on appelle un arbre de jeu.

Dans cette optique, et pour l'accomplissement de notre projet, nous avons utilisé l'algorithme de recherche qui est décrit dans le premier chapitre « *Introduction à la théorie des jeux* » Fail-Soft Alpha-Bêta.

5. Conception détaillée :

Cette partie traite de l'application des algorithmes évolutionnaires sur les réseaux de neurones pour la création de nouvelles stratégies de jeu.

5.1. Algorithme général du processus d'évolution :

L'algorithme qui génère le processus illustré dans les sections précédentes est donné par le pseudo-code suivant :

Initialisation :

- Choisir 15 Stratégies (réseaux de neurones) en utilisant la distribution uniforme dans l'intervalle [-0.2, 0.2] pour déterminer les poids de chaque stratégie (5046 poids), les stratégies sont notées pi, i=1..15 et la valeur du roi est initialement fixée à 2.0
- Chaque stratégie à un vecteur adaptatif σi, i=1..15 chaque élément de ce vecteur correspond à un poids ou seuil et serve à contrôler le pas pour une nouvelle mutation (chaque vecteur se compose de 5046 éléments).
- Initialiser les éléments de ces 15 vecteurs à 0.05

Déroulement :

Répéter

Pour chaque stratégie initialiser leurs points totaux de jeu à 0 ;

Pour chaque stratégie choisir aléatoirement 5 adversaires ;

Pour chaque adversaire commencer le jeu avec lui (le joueur).

Comment jouer :

Le jeu se déroule en utilisant l'algorithme de recherche Fail-Soft Alpha-Beta.

Répéter

```
Mettre profondeur à 4 ;
int alphabêta(int profondeur, int alpha, int bêta)
{
   Si (jeu est terminé ou profondeur <= 0)
      retourner score résultant ou eval();
      mouvement MeilleurMouvement ;
   int current = -INFINI;
   Pour chaque (mouvement m) {
      Faire le mouvement m;
      int score = - alphabêta(profondeur - 1, -bêta, -alpha)
      Retirer le mouvement m; // C'est pour libérer l'espace
      mémoire
      Si (score >= current) {
         current = score;
         MeilleurMouvement = m;
         Si (score >= alpha){
                alpha = score;
```

```
            MeilleurMouvement = m ;
            Si (score >= bêta)
               break;
         }
      }
   }
   return current;
}  // Fin de l'alpha-bêta fail-soft
```

Remarque :

« Mouvement » signifie d'établir s'il y a de mouvement obligatoire, si c'est le cas déterminer le maximum de mouvements successifs obligatoires et les organiser dans une liste et augmenter la profondeur par le plus petit chiffre paire supérieur ou égale le nombre de mouvements successifs, sinon (c'est-à-dire il n'y a pas de mouvement obligatoire) alors établir les mouvements normaux.

Effectuer le meilleur mouvement qui entraîne le changement de la table, donc reste la réponse de l'adversaire.

```
// Réponse de l'adversaire
Mettre profondeur à 4 ;
int alphabêta(int profondeur, int alpha, int bêta)
{
   Si (jeu est terminé ou profondeur <= 0)
      retourner score résultant ou eval(); // Entrer la table dans
                                              le RNA pour l'évaluation
   mouvement MeilleurMouvement ;
   int current = -INFINI;
   Pour chaque (mouvement m) {
      Faire le mouvement m;
      int score = - alphabêta(profondeur - 1, -bêta, -alpha)
      Retirer le mouvement m; // pour libérer l'espace mémoire
      Si (score >= current) {
         current = score;
         MeilleurMouvement = m;
         Si (score >= alpha){
```

```
                alpha = score;
                MeilleurMouvement = m ;
                Si (score >= bêta)
                    break;
               }
            }
        }
        return current;
    }
    Until jeu est terminé
  Fin pour
Fin pour
```

- Classer les différentes stratégies selon leur gain dans les différentes parties jouées.
- Sélectionner les 15 premières stratégies.
- **Pour chaque** stratégie créée une progéniture en appliquant l'opérateur de mutation selon les règles décrites précédemment :

  $\sigma'_i(j) = \sigma_i(j) \cdot Exp(\tau \cdot Nj(0, 1)), j=1,\ldots, Nw.$

  $W'_i(j) = W_i(j) + \sigma'_i(j)Nj(0,1), j=1,\ldots, Nw.$

Jusqu'à obtenir une meilleure stratégie.

Algorithme 4.1 : Algorithme général de l'application.

5.2. Déroulement de l'algorithme :

À chaque génération, un joueur va être définit par leur réseau de neurones associé et qui représente ainsi une stratégie de jeu.

***Remarque* :** tous les réseaux de neurones utilisés ont la même structure générale décrit dans la (figure 4.6), et dans lesquels tous les poids de connexions, les seuils et la valeur de Roi, sont évolués.

Une population de 15 stratégies (réseaux de neurones), **Pi ; i=1…15**, définie par les **poids** et les **seuils** pour chaque réseau de neurones et la valeur associée de la stratégie de **k**, est créée au hasard (figure 4.7).

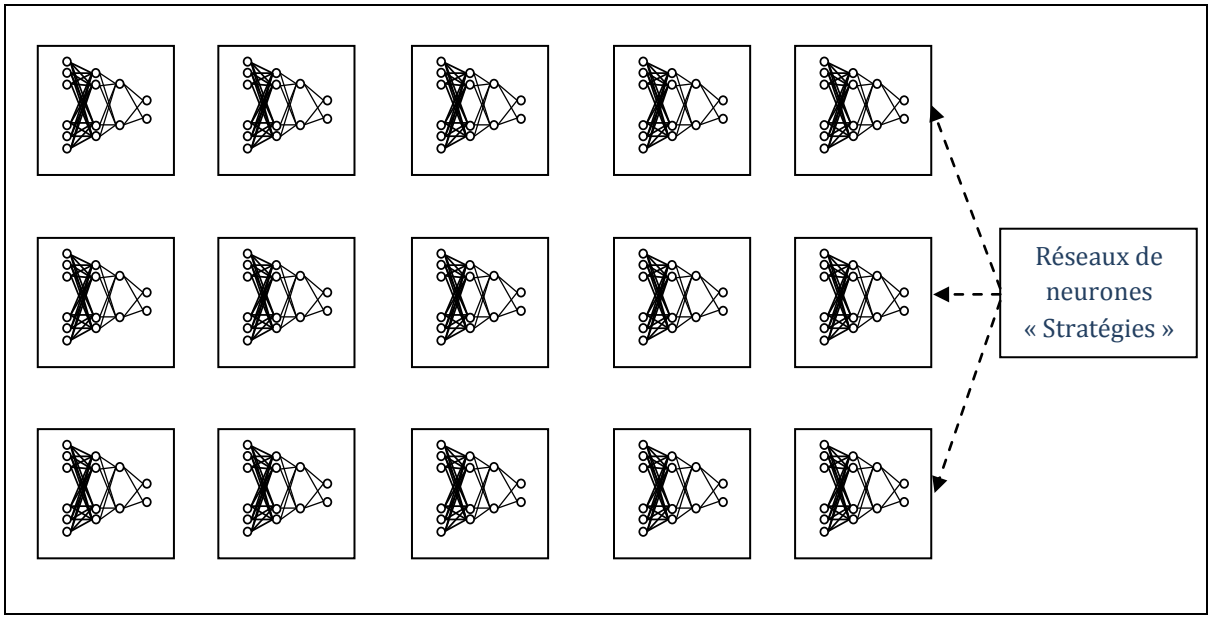


Figure 4.7: Population initiale contient 15 Réseaux de neurones

Les poids et les seuils sont générés en utilisant la distribution uniforme entre **[-0.2, 2.0]**, avec la valeur initiale de l'ensemble **k** est **2**.

Chaque stratégie a eu un vecteur de paramètre auto-adaptatif associé **σi, i=1,…,15** où chaque composant a correspondu à un poids ou à un seuil et a servi à contrôler la taille de l'étape de la recherche de nouveaux paramètres mutés du réseau de neurones.

Pour être conformé à l'intervalle de l'initialisation, les paramètres auto-adaptatifs pour les poids et les seuils ont été initialisés à **0.05.**

Chaque parent a généré une stratégie progéniture (fils) en changeant tous les poids et les seuils associés (figure 4.8), et probablement la valeur de **k** aussi bien. Spécifiquement, pour chaque parent **Pi, i=1,...,15** un fils **P'i, i=1,…,15** a été crée par :

$$\begin{cases} \sigma'i(j) = \sigma i(j) \,.\, Exp\,(\tau \,.\, Nj(0,\,1)),\ j=1,\ldots,\ Nw. \\ W'\ i(j) = W\ i(j) + \sigma'i(j) Nj(0,1),\ j=1,\ldots,\ Nw. \end{cases}$$

Où :

- **Nw:** est le nombre de poids et de seuils dans le réseau de neurones (ici c'est **5046**).
- **τ = 1/sqrt (2sqrt (Nw))= 0.0839.**

- **Nj(0,1) :** est une variable aléatoire gaussienne standard recalculé pour chaque **j.**
- La valeur **k'** du Roi fils a été obtenue par : **K'i = Ki + d**, Où **"d"** a été choisi uniformément au hasard de **{-0.1, 0, 0.1}.**
- Pour la convenance, la valeur de **k'j** se situe dans l'intervalle **[1.0, 3.0]**.

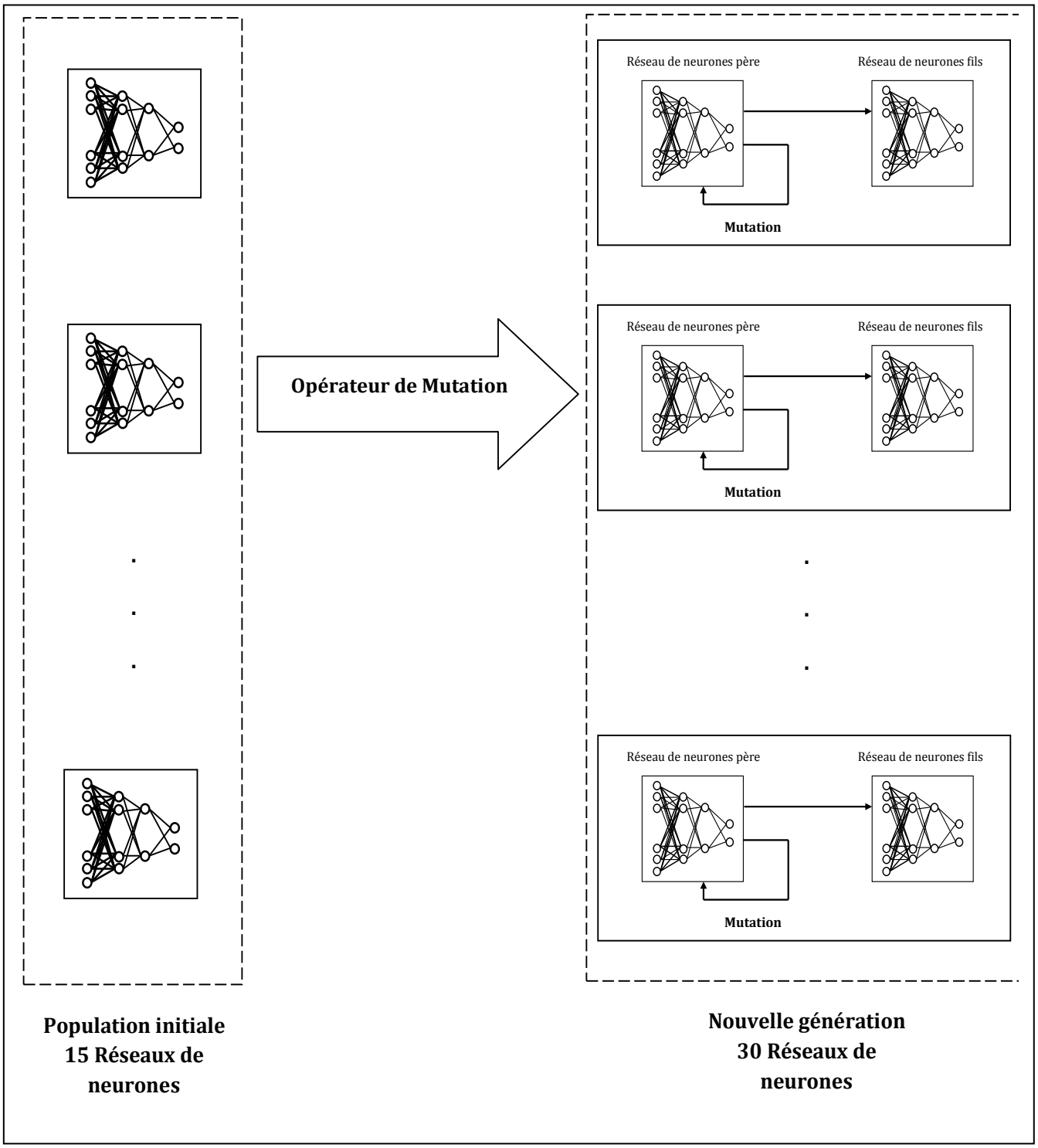


Figure 4.8 : Application de l'opérateur de Mutation sur une population de 15 réseaux de neurones

Tous les parents et leurs fils ont concurrencés pour la survie en jouant des jeux de dames et en recevant des points pour leurs résultats du jeu. Chaque joueur a joué un jeu contre chacun des cinq adversaires qui sont choisis aléatoirement de la population (figure 4.9).

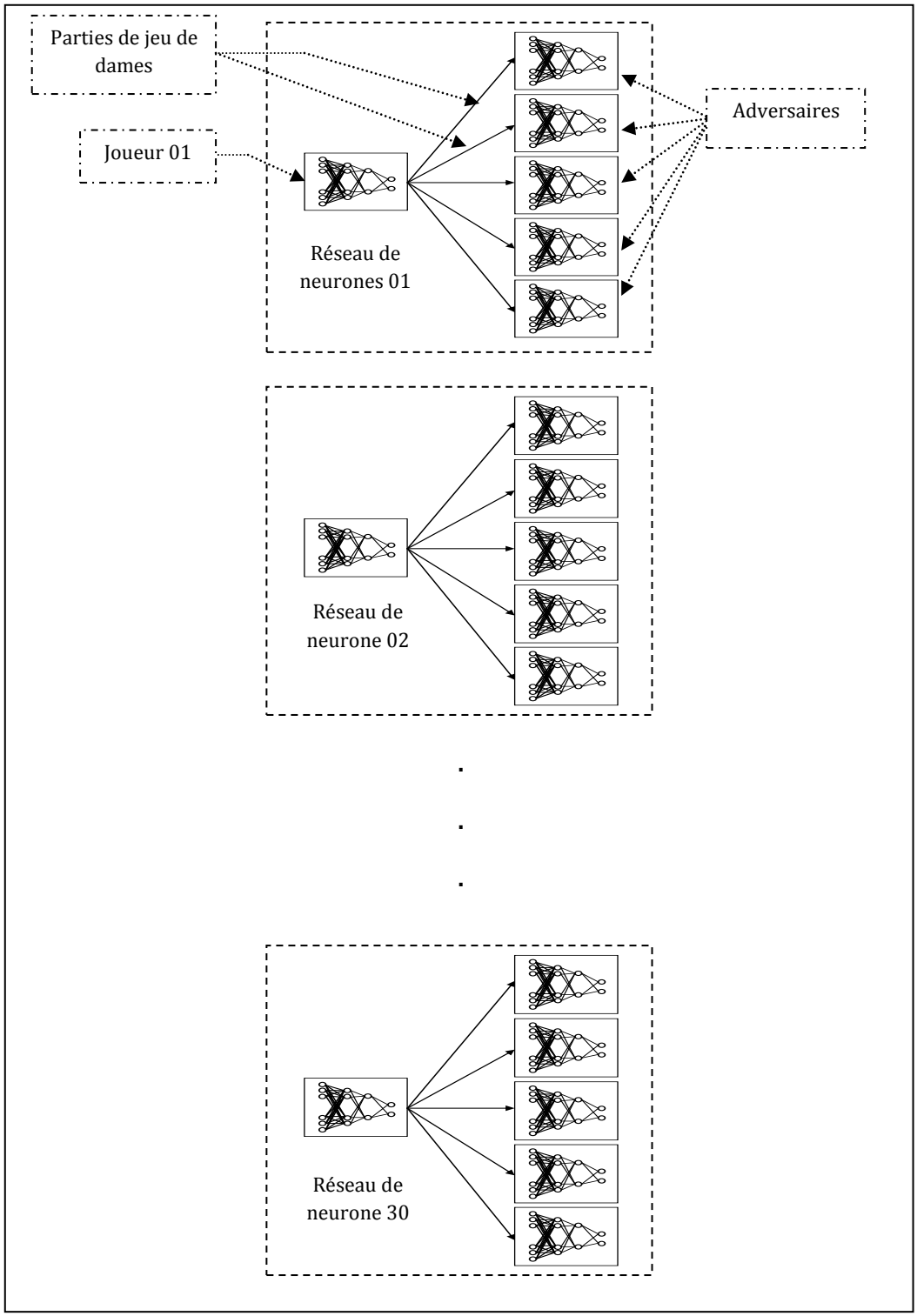


Figure 4.9 : Chaque réseau de neurones joue une partie de jeu de dame contre 5 autres réseaux de neurones de la même génération.

Dans chacun de ces cinq jeux, le joueur a toujours joué avec les pièces noires, tandis que l'adversaire qui a été choisis aléatoirement a toujours joué avec les pièces blanches. Dans chaque jeu, le joueur a marqué -2, 0, ou +1 Points selon s'il a **perdu**, faire une partie **nulle**, ou a **gagné** le jeu, respectivement.

***Remarque* :** Une partie est déclarée nulle après 100 mouvements pour chaque coté.

De même chacun des adversaires a également marqué -2, 0, ou -1 point selon les résultats. Ces valeurs étaient quelque peu arbitraires, mais elles ont reflété un protocole généralement raisonnable pour avoir une perte soit deux fois plus coûteuse qu'une victoire. Au total il y avait 150 jeux par génération, avec chaque stratégie participant à une moyenne de 10 jeux.

Après que tous les jeux aient été terminés, les 15 stratégies qui ont reçu le plus grand total des points ont été maintenues et deviendront que les parents de la prochaine génération. Ensuite, le processus d'évolution sera réitéré avec cette nouvelle génération.

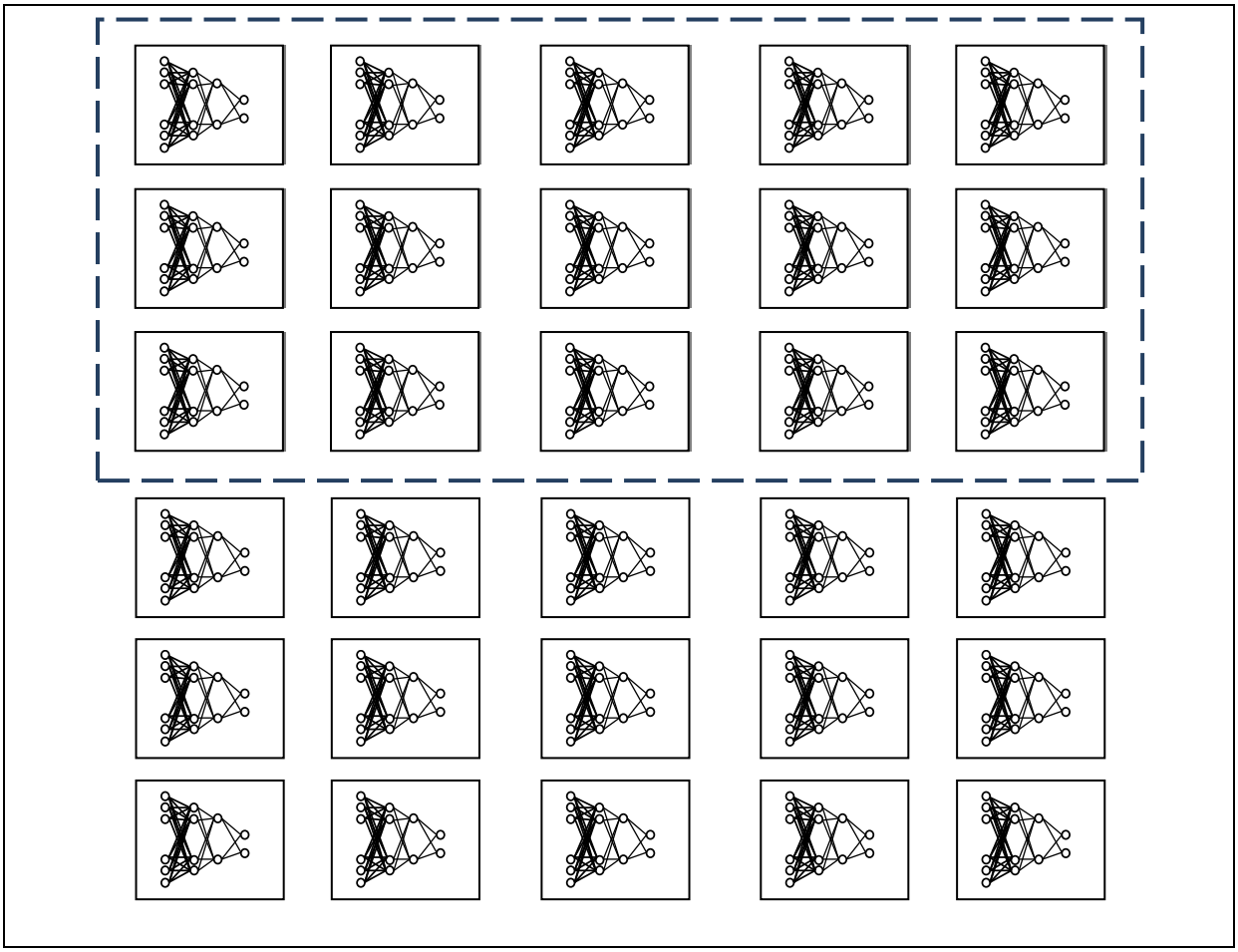


Figure 4.10 : Application de l'opérateur de sélection sur les 15 premiers réseaux de neurones

6. Modélisation :

La motivation fondamentale de la modélisation est de fournir une démarche antérieure afin de réduire la complexité du système étudié lors de la conception et d'organiser la réalisation du projet en définissant les modules et les étapes de la réalisation. Plusieurs démarches de modélisation sont utilisées. Nous adoptons dans notre travail une approche objet basée sur un outil de modélisation *UML*.

6.1. Présentation de l'UML :

Le langage *UML* **«** *Unified Modeling Language* » est un langage de modélisation qui a pour but de faciliter les transitions, lors du développement d'un projet. Il permet de structurer un projet et de le matérialiser graphiquement sous forme de diagrammes compréhensibles par les non informaticiens. Aucune connaissance de langage informatique n'est pré-requise.

Cette modélisation permet dans un second temps de développer le code informatique, le plus souvent à l'aide d'un langage orienté objet. La description de projets en UML est une étape nécessaire qui permet de gagner beaucoup de temps dans le développement d'une application car la mise au point du code en est moins fastidieuse et le risque d'erreurs de conception ou de réalisation est plus limité. Bien que conçue pour la gestion de projets de grande envergure, l'utilisation de cette méthodologie est bénéfique même pour les projets les plus modestes.

6.2. Historique : [47]

En 1994, naissance de l'UML chez Rational Software Corporation à l'initiative de G. Booch et de J. Rumbaugh, il est le fruit d'un travail d'unification et de standardisation de trois méthodes de modélisation dominantes développées dans les années 90 : OMT, Booch et OOSE.

En 1997, UML 1.1 a été standardisé par l'OMG (Object Management Group), suite à la demande émanant de la collaboration de plusieurs entreprises (Hewlett-Packard, IBM, i-Logix, ICON Computing, IntelliCorp, MCI Systemhouse, Microsoft, ObjecTime, Oracle, Platinum Technology, Ptech, Rational Software Corporation, Reich Technologies, Softeam, Sterling Software, Taskon et Unisys).

Depuis 1999, la version actuelle est UML **1.3** (la version 1.4 sera bientôt prête, afin de préparer la prochaine version 2.0).

6.3. Objectifs de l'UML : [48]

Au final, le langage UML est une synthèse de tous les concepts et les formalismes méthodologiques les plus utilisés, pouvant être utilisé, grâce à sa simplicité et ça son universalité, comme langage de modélisation pour la plupart des systèmes ils nécessiteraient le développement.

Le langage UML permet ainsi d'apporter des solutions lors du développement des systèmes informatisé :

- Décomposer le processus de développement en distinguant la phase d'analyse (aspects fonctionnels) de la phase de réalisation (aspects technologiques et architecturaux).
- Décomposer le système en sous-systèmes plus facilement abordables : réduction de la complexité, répartition du travail, réutilisation des sous-systèmes.
- Utiliser une technologie de haut niveau proche de la réalité pour aborder le développement.

6.4. Les différentes vues d'UML : [49]

La modélisation proposée par le langage UML se réalise principalement sous forme graphique, en usant de divers types de diagrammes spécifiques, répartis en trois groupes :

- **Vue fonctionnelle :**

Interactive, qui est représentée à l'aide de *diagrammes de cas d'utilisation*, *diagrammes des séquences, et* les *diagrammes de collaboration*. Elle cherche à appréhender les interactions entre les différents acteurs/utilisateurs et le système, sous forme d'objectif à atteindre d'un côté et sous forme chronologique de scénarios d'interaction typiques de l'autre.

- **Vue structurelle :**

Appelée aussi *statique*, réunit les *diagrammes de classes* et les *diagrammes de packages*. Les premiers favorisent la structuration des données et tentent d'identifier les objets/composants constituant le programme, leurs attributs, opérations et méthodes, ainsi que les liens ou associations qui les unissent. Les seconds s'attachent à regrouper les classes fortement liées entre elles en des composants les plus autonomes possibles. A l'intérieur de chaque package, on trouve un diagramme de classes.

- **Vue dynamique :**

Qui est exprimée par les *diagrammes d'états*. Cette vue est plus algorithmique et orientée « traitement», elle vise à décrire l'évolution (la dynamique) des objets complexes du

programme tout au long de leur cycle de vie. De leur naissance à leur mort, les objets voient leurs changements d'états guidés par les interactions avec les autres objets. Le *diagramme d'activité* est une sorte d'organigramme correspondant à une version simplifiée du diagramme d'états. Il permet de modéliser des activités qui se déroulent en parallèle les unes des autres, quand ce parallélisme peut poser problème. En général, les diagrammes d'états à eux seuls ne permettent pas de faire apparaître les problèmes spécifiques posés par la synchronisation des processus en concurrence, pour assurer la cohérence du comportement et l'absence d'inter-blocage. Etablir un diagramme d'activité peut aider à mettre au point un diagramme d'états.

Certains de ces diagrammes sont indépendants, alors que d'autre servent de base de travail ou bien sont la continuité d'autres diagrammes.

Afin de développer notre application, on s'intéresse aux diagrammes suivants :

- **Diagramme de cas d'utilisation :**

Un cas d'utilisation (use case) modélise une interaction entre le système informatique à développer et un utilisateur ou acteur interagissant avec le système. Plus précisément, un cas d'utilisation décrit une séquence d'actions réalisées par le système qui produit un résultat observable pour un acteur.

- **Diagramme des classes :**

Un diagramme des classes décrit le type des objets ou données du système ainsi que les différentes formes de relation statiques qui les relient entre eux. Le diagramme de classes qui est unique, se construit en partie à l'aide des informations issues des différents de séquence. Il permet d'obtenir le squelette du code par génération automatique de code ; il s'agit donc de la dernière étape d'analyse juste avant le codage proprement dit.

- **Diagramme de séquence :**

Les diagrammes de séquences mettent en valeur les échanges de messages (déclenchant des événements) entre acteurs et objets (ou entre objets et objets) de manière chronologique, l'évolution du temps se lisant de haut en bas. Un diagramme de séquences est un moyen semi-formel de capturer le comportement de tous les objets et acteurs impliqués dans un cas d'utilisation. On peut indiquer un type de message particulier : les retours de fonction qui, bien entendu, ne concernent aucun message mais signifient la fin de l'appel de l'objet appelé. Ils permettent d'indiquer la libération de l'objet appelant (ou de l'acteur). Un emploi abusif de retours de fonction peut alourdir considérablement le diagramme, aussi un usage parcimonieux est-il conseillé.

4.7. **Présentation des diagrammes :**

Dans la section suivante, nous allons identifier les trois diagrammes illustrés précédemment afin de mettre en œuvre l'approche choisie pour le développement de notre système :

CHAPITRE V

IMPLÉMENTATION

Après avoir détaillé la conception de notre application, nous allons présenter dans ce chapitre la partie de réalisation ainsi que les déférents outils pour le développement du logiciel.

1. Introduction :

Dans ce chapitre, nous présentons l'environnement sur lequel nous avons développé notre application, les différents outils utilisés ainsi que les composantes applicatives réalisées. Enfin nous présentons les principales interfaces et fenêtres de l'application.

2. Présentation du logiciel « *Game strategy programing* » :

Comme l'indique son nom, notre logiciel a pour objectifs d'évaluer des stratégies de jeu combinatoire (jeu de dames) on se basant sur l'idée d'hybridation des réseaux de neurones avec les algorithmes évolutionnaires. La programmation de ces stratégies se fait par le déroulement d'un processus évolutionnaire qui contient au début un ensemble de stratégies créées aléatoirement. Ce logiciel permet de créer de nouvelles générations, de les importer et de les enregistrer, parce que le parcours d'une seule génération devient très long, à titre exemple une configuration par défaut nécessite presque 5 heures.

3. Outils de développement :

3.1. Environnement matériel de développement :

Afin de réaliser notre projet, nous avons utilisé un pc portable ayant les caractéristiques suivantes :

- Fabriquant : Lenovo
- Modèle : G50-30
- Processeur : Intel ® Pentium ® CPU B970 @ 2,30 GHz 2,30 GHz
- Mémoire installée (RAM) : 4,00 Go
- Type du système : Système d'exploitation 64 bits, Microsoft Windows 10.

3.2. Environnement logiciel de développement :

Après avoir passé par l'étape de l'analyse et conception, il nous reste de choisir une plateforme de développement appropriée pour mettre en place tous les efforts qu'on a fournit tout au long de ce thème. Pour la mise en œuvre de ce prototype, nous avons choisi l'environnement de développement Borland Delphi.

Delphi est un système de développement visuel rapide sous Windows (***R**apid **A**pplication **D**evelopment)* qui utilise le langage Pascal Orienté Objet et permet de créer des applications fenêtrées directement exécutables et redistribuables librement sous Windows ou DOS, avec un minimum de programmation.

Ce langage est facile à apprendre car, les objets utilisés ont des propriétés et des méthodes, les propriétés sont les caractéristiques de l'objet (couleur, taille, ...) tandis que les méthodes sont les procédures (classiques ou événementielles) et fonctions qui y sont rattachées.

Nous avons choisi la version 7 de Delphi car elle fournit tous les outils nécessaires pour développer, tester et déployer des applications, notamment : une importante bibliothèque de composants réutilisables, une suite d'outils de conception, de modèles d'applications, de fiches et d'experts de programmation que les versions précédentes du logiciel ne possédaient pas.

4. Aperçus sur les classes utilisées :

Dans cette section nous allons présenter des méthodes de classes les plus importantes. On commence par la classe ***TTournement***, elle effectue la gestion du tournoi pour cela elle contient des méthodes pour accepter de nouveaux joueurs, choisir des adversaires, classer les joueurs selon leurs gain et sélectionner les meilleures entre eux.

La classe ***TPlayer***, elle détermine les coups possibles, exécuter un coup en changeant la table du jeu, effectuer le choix de meilleur coup en utilisant l'algorithme de recherche Fail-soft Alpha-Beta.

La classe ***TStrategy*** implémente la structure du réseau de neurones, elle permet d'initialiser les stratégies, générer une stratégie fille selon les lois décrites dans le chapitre précédent, et effectuer l'évaluation de la table du jeu.

Enfin, la classe ***TTable*** effectue la gestion de la table du jeu.

5. Présentation de quelques méthodes de classe :

La création de notre application a nécessité le passage par plusieurs méthodes, nous allons présenter les plus importantes :

La méthode ***PlayerClassification*** de la classe ***TTournement,*** permet de classer les joueurs selon leur résultat obtenue. Le pseudo code qui décrit cette méthode est décrit come suit :

```
Procedure TTournement.PlayersClassification;
var
i,j,indmin:byte;
p,p1:Tplayer;
begin
for i:=1 to members.Count-1 do
```

```
 begin
 p:=members.items[i-1];
 indmin:=i;
 for j:=i+1 to members.Count do
  begin
  p1:= members.Items[j-1];
  if p1.Result<p.Result then
   begin
   p:=p1;
   indmin:=j;
   end;
  end;
 p1:=members.items[i-1];
 members.items[i-1]:=p1;
 members.items[indmin-1]:=p1
 end;
for i:= 1 to members.Count do
 begin
 p:=members.items[i-1];
 p.Number:=i;
 end;
end;
```

La méthode ***AlphaBetaFailSoft*** de la classe ***TPlayer***, cette méthode est utilisée au cour du déroulement de la compétition pour effectuer la recherche du meilleur coup. Le pseudo code qui décrit cette méthode est le suivant :

```
Procedure TPlayer.AlphaBetaFailSoft(Table      :TTable;
                                    Adversaire :TPlayer;
                                    BestMove   :TMove;
                                    ply        :Byte;
                                    Alpha      :extended;
                                    Beta       :extended;
                                    var
                                    score1     :extended;
```

```
                                        Tour        :Integer;
                                        Result      :Integer );

var
current,score  :extended;
pile           :tpile;
table1         :ttable;
move           :tmove;
ad,pl          :tplayer;
k2,k           :integer;
tab            :array[1..14]of byte;
tabs           :tabstracttable;
h:integer;
begin
 k:=0;
 pile:=tpile.Create;
 DetectAllMoves(table,pile);
 inc(itera);
 arrival:=arrival+1;
 if pile.Count=0 then
   score1:=-1
else
  begin
       if ply<= 0 then
         begin
               tabs:=GetAbstractTable(table,adversaire);
               strategy.SetVector(tabs);
               score1:=Strategy.EvaluateStrategy;
         end
       else
         begin
               current:=-1;
               table1:=ttable.create;
               pl:=tplayer.PseudoCreate;
               ad:=tplayer.PseudoCreate;
               while pile.Count>0 do
               begin
```

```
                    table.CopyTo(table1);
                    adversaire.CopyTo(ad);
                    self.CopyTo(pl);
                    move:= pile.Depiler;
                    pl.ExecuteMove(table1,ad,move);
                    ad.AlphaBetaFailSoft(table1,pl,bestmove,ply-

                    1,-beta,-alpha,score1,tour,result);
                     score:=-score1;
                    if score>= current then
                      begin
                           current:=score;
                           k:=move.count;
                            for k2:=1 to move.count do
                             begin
                             tab[k2]:=move.first;
                             move.deletef;
                             end;
                          if score>=alpha then
                             begin
                                  alpha:=score;
                               if score>=beta then
                                  break;
                             end;
                      end;
                  if assigned(move) then
                    move.Free;
               end;
                table1.free;
                ad.PsaudoDestroy;
                pl.PsaudoDestroy;
                score1:=current;
        end;
 end;
pile.free;
dec(itera);
bestmove.clear;
```

```
   for k2:=1 to k do
    begin
     bestmove.add(tab[k2]);
    end;
end;
```

La méthode ***InitialiseToPlay*** de la classe ***TTable***, est chargée d'initialiser la table du jeu de dames à chaque nouvelle partie. Son pseudo code est décrit comme suit :

```
Procedure  TTable.InitialiseToPlay;
var
i:byte;
begin
  for i:=1 to 32 do
    begin
       if i<=12 then
        table[i]:=i
       else
          begin
          if (i>=21) then
           begin
            table[i]:=-(33-i);
           end
            else
            table[i]:=0;
          end;
    end;
end;
```

La méthode ***EvaluateStrategy*** qui appartient à la classe ***TStrategy***, effectue une évaluation de la table du jeu

```
function Tstrategy.EvaluateStrategy:Extended;
var Layer:Tlayer;
    Neural:TNeural;
begin
NeuralNetwork.EvaluateNeuralNetwork;
Layer:=NeuralNetwork.ListLayer.Items[NeuralNetwork.ListLayer.Count-
1];    Neural:=Layer.ListNeural.Items[0];
EvaluateStrategy:=Neural.Value;
end;
```

6. Création et intégration des interfaces :

Voici l'enchaînement de quelques interfaces accompagnées par leurs scénarios descriptifs :

6.1. Interface d'accueil :

C'est la première fenêtre qui s'affiche pour les utilisateurs. Pour créer une nouvelle génération, cliquer sur nouveau dans le Menu « **Fichier »**, vous pouvez aussi utiliser le bouton « **Nouveau »** du bar d'outils, un volet sera apparait, il contient une liste hiérarchique d'élément représentant l'ensemble des paramètres du processus évolutionnaire, comme le montrer la figure suivante.

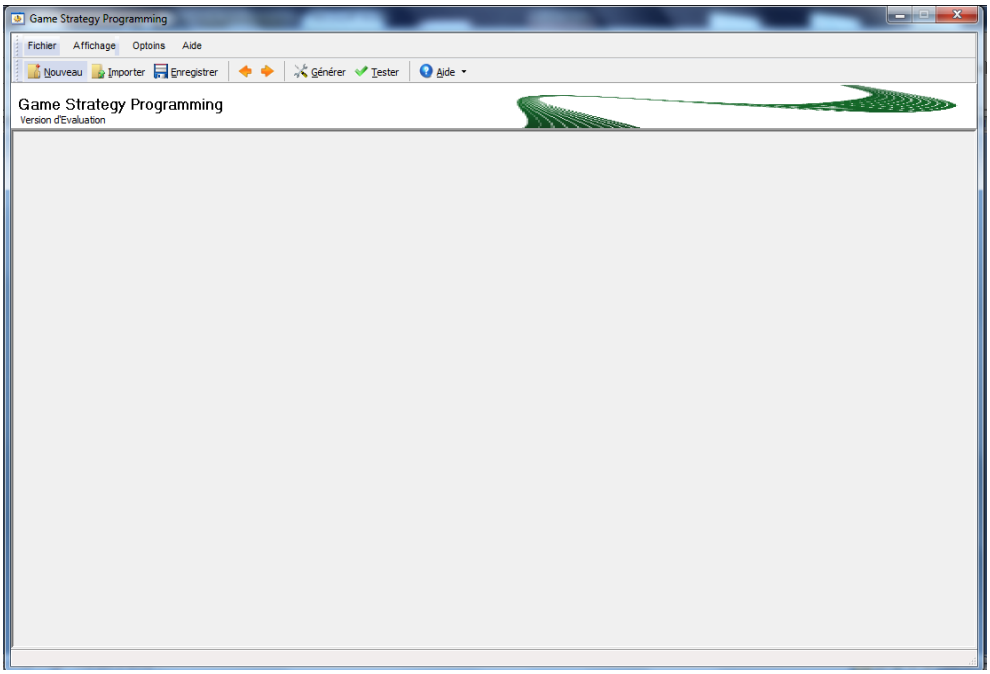


Figure 5.1 : Interface d'accueil.

6.2. Interface de configuration des paramètres de la génération :

Les éléments qui représentent l'ensemble des paramètres d'évolution doivent être configuré les uns après les autres, en respectant l'ordre dont ils sont arrangés parce qu'il existe certains paramètres qui devront configurés avant des autres. Donc pour configurer les paramètres de la génération, on procède les étapes décrites dans ce qui suit.

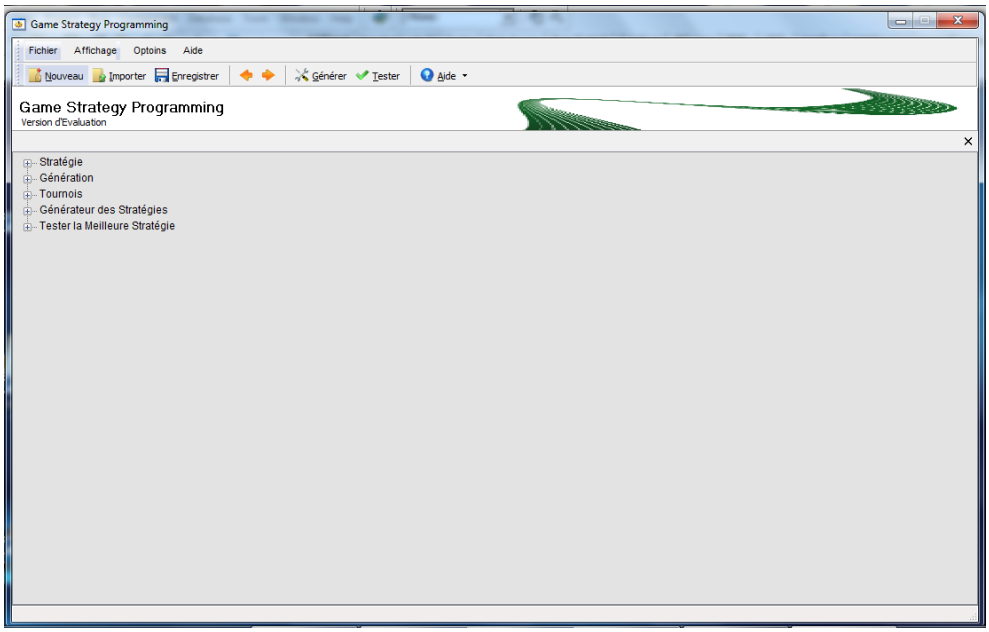


Figure 5.2 : Interface de configuration des paramètres de la génération.

6.3. Interface « nouvelle stratégie » :

C'est la première étape, on doit paramétrer la stratégie qui est constitué de deux sous éléments :

- Un réseau de neurones, dont sa configuration restera la plus importante, car il constitue le noyau du développement stratégique du jeu. Il possède des poids et des polarisations (seuils) par les quelles on conserve les différentes évaluations. Ainsi, pour le jeu de dames, nous avons choisie une structure spatiale identique à celle du réseau de neurones défini dans le chapitre précédent.
- Les valeurs des pièces, pour le jeu de dames il existe deux types de pièces, normale et roi, la première ne change pas dans sa configuration initiale, par contre la pièce d'un roi devrait être changées dans le processus de génération. Par défaut, la pièce normale vaut 1 pendant que la pièce roi vaut 3.

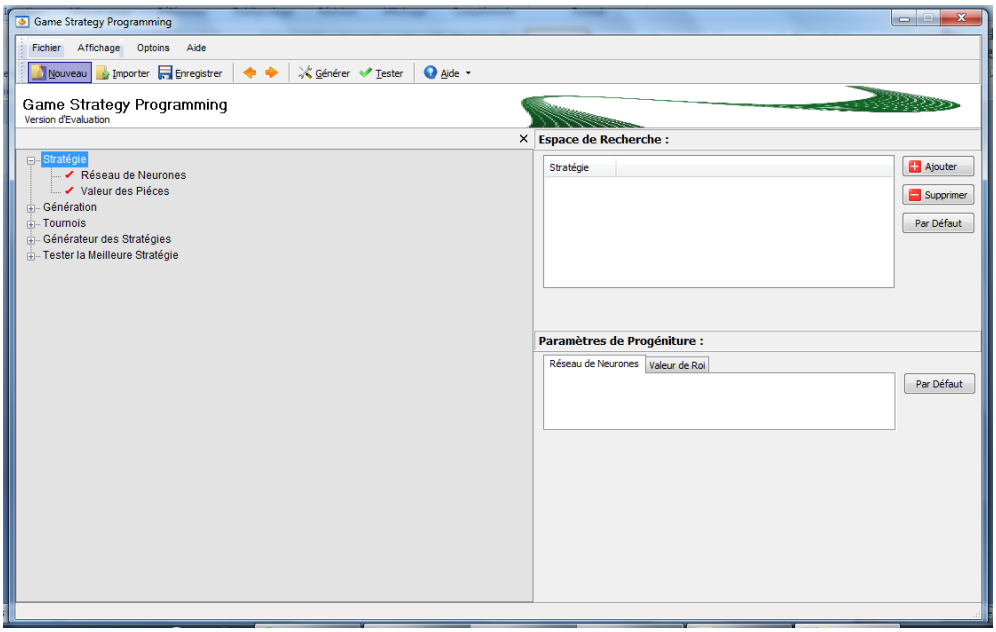


Figure 5.3 : Interface de création de nouvelle stratégie.

6.4. Interface « Génération » :

C'est la deuxième étape, elle est constituée de :

- Un espace de recherche qui contient l'ensemble des différentes stratégies où chaque stratégie dispose sa propre configuration (réseau de neurones et valeur des pièces).
- Des paramètres de progénitures avec lesquels des nouvelles stratégies sont obtenus par l'opération de progéniture qui est appliquée à chaque stratégie de l'espace de recherche.

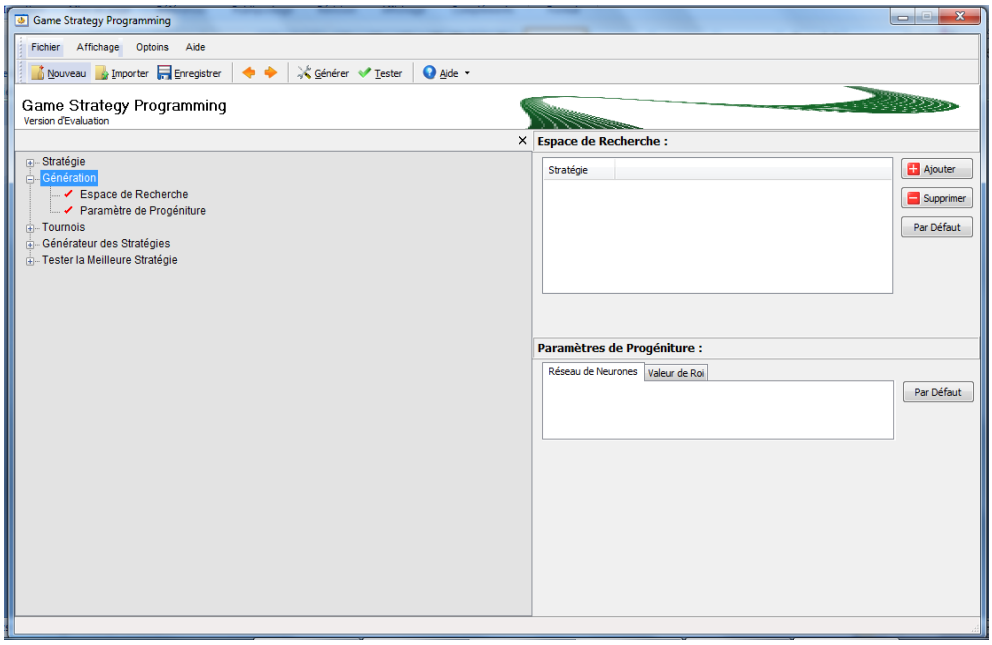


Figure 5.4 : Interface de création de nouvelle stratégie.

6.5. Interface « Tournois » :

La troisième étape consiste en un tournoi qui est composé d'un ensemble de joueurs, chaque joueur joue avec sa stratégie correspondante dans l'espace de recherche. Cette correspondance nous admet de calculer la fonction d'évaluation de chaque stratégie. La fonction d'évaluation d'une stratégie, représente la somme des différents résultats obtenus.

Il est tout à fait claire que le nombre de joueur total égale au nombre de stratégies plus les stratégies de progénitures. La configuration de cet élément se fait comme suit :

- Pour le jeu on spécifie le nombre de mouvements et de coûts possible dans une partie de jeu, ainsi le résultat d'une partie et la profondeur de recherche.
- Pour le tournoi on effectue pour chaque joueur le nombre d'adversaire et le nombre de joueurs à sélectionner dans les prochaines générations.

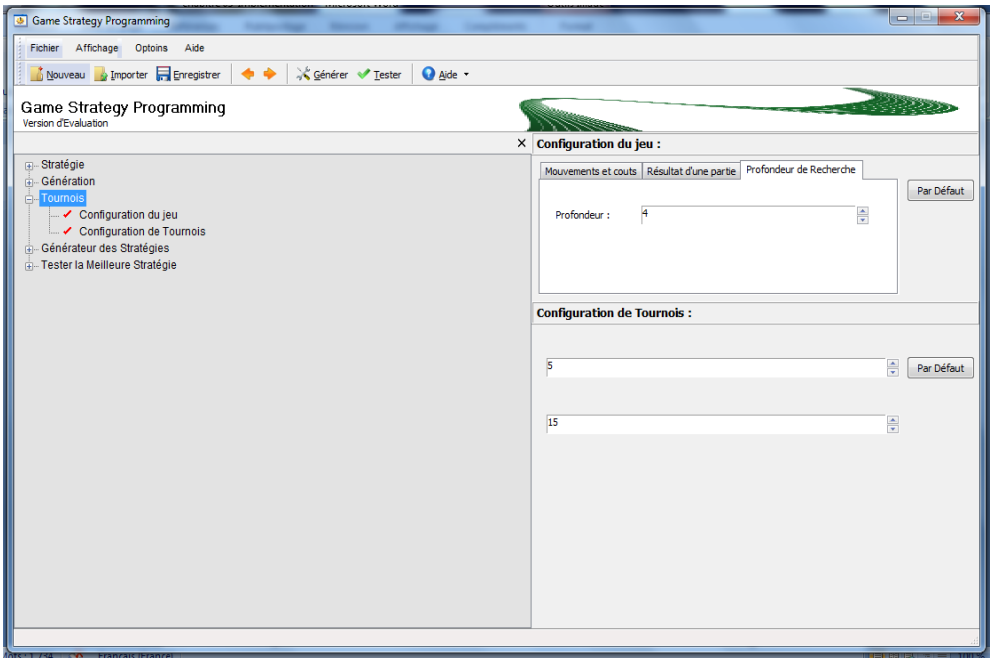


Figure 5.5 : Interface de tournois.

6.6. Interface « Exécuter le générateur du processus évolutionnaire » :

La quatrième étape, après avoir configuré les différents paramètres précédents, nous arrivons à générer notre algorithme évolutionnaire, ceci se fait en cliquant sur le bouton « **Générer »** dans la barre d'outils, un volet sera affiché contenant la plupart des informations qui assurent une simulation visuelle du processus de génération. Pour débuter ce processus cliquez sur le bouton « **Démarrer »**.

- L'onglet « **Général »** vous permet de contrôler l'exécution des étapes successives du processus évolutionnaire (Initialisation, Progéniture, Evaluation, Sélection et Mise à jour). Il faut noter Que l'étape de l'initialisation n'est exécutée que dans la première génération.
- L'onglet « **Détail »** permet de visualiser l'évaluation d'une génération, ou les différents joueurs entrent en concurrence entre eux.

Vous pouvez à un instant donné arrêter l'exécution du processus de génération en cliquant sur le bouton **« Stop »**, ce qui permet de tester la meilleure stratégie obtenue.

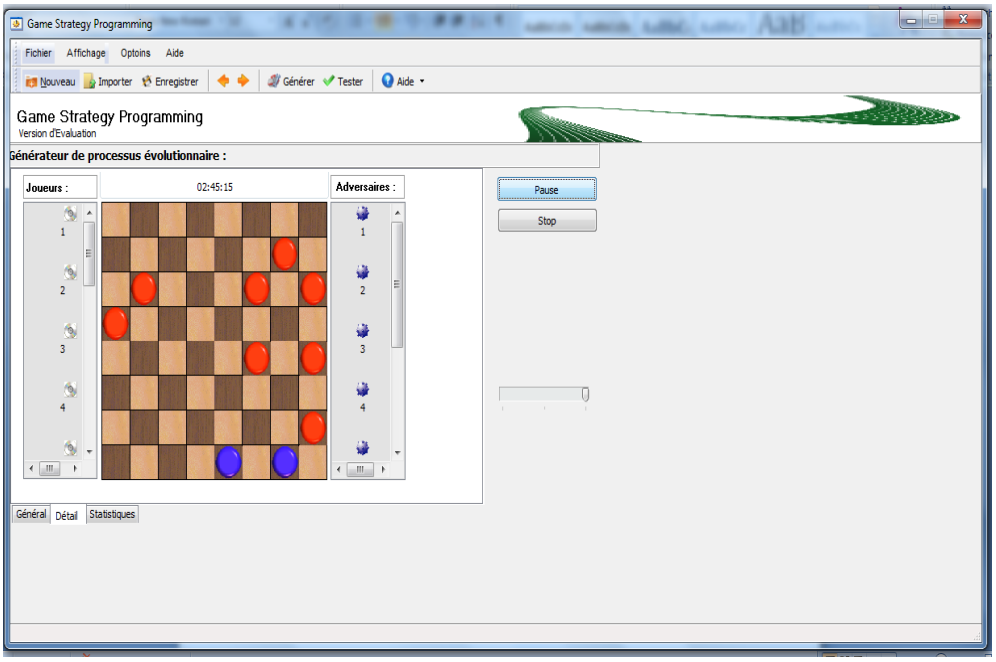


Figure 5.6 : Interface d'exécution du générateur du processus évolutionnaire.

6.7. Interface « Exécuter le test » :

Enfin, la dernière tâche qu'on doit effectuer est de tester la meilleure stratégie obtenue à partir de plusieurs générations successives, en jouant un jeu de dames contre l'ordinateur qui utilise cette stratégie. Pour cela, cliquez sur le bouton **« Tester »** dans la barre d'outils. Une fenêtre s'affiche, cliquez ensuite sur **« Nouvelle partie »** pour commencer un nouveau jeu.

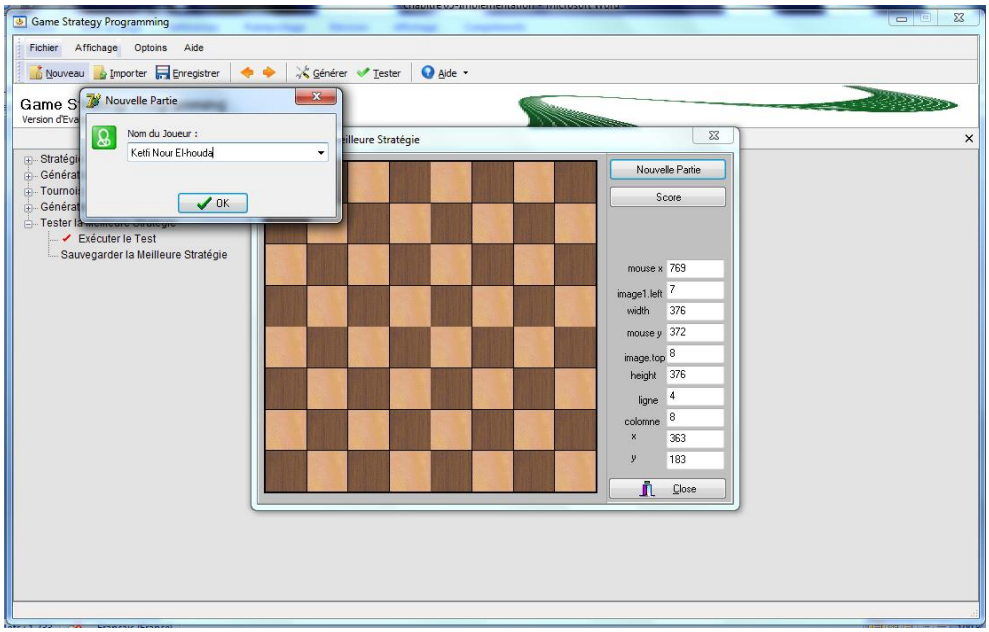


Figure 5.7 : Interface d'exécution du test.

7. Conclusion :

Nous avons vu dans ce dernier chapitre que la réalisation d'un logiciel qui satisfait les besoins et les objectifs décrits dans ce travail impliquent d'abord une compréhension approfondie du problème à traiter. Par la suite, il faut établir un plan de travail qui trace les points nécessaires conduisant à une meilleure solution. Enfin, une bonne modélisation du problème donne lieu à une facilité d'implémentation ce qui permet d'aboutir à un logiciel fiable.

5. Conclusion :

Combiner les différents types de diagrammes offrent une vue complète des aspects statiques et dynamiques des systèmes. Comme nous pouvons le constater, l'activité de la conception a facilité la compréhension de notre système, qui ébauche vers l'activité d'implémentation.

Dans le chapitre suivant nous allons présenter l'implémentation et la réalisation de notre application. Pour montrer l'efficacité de l'approche illustrée dans les sections précédentes, on a besoin de réaliser un logiciel sous forme d'un prototype pédagogique qui permet d'évoluer un jeu combinatoire (jeu de dames) selon les points abordés dans ce présent chapitre.

Bibliographie **:**

[5]. Kumar Chellapilla, David B. Fogel "*Evolving an Expert Checkers Playing Program without Using Human Expertise*"

[6]. Fatiha Kacher & Karima Bouibed "*La théorie des jeux*", 2012.

[11]. Tuomas W. Sandholm and Robert H. Crites. "*Multiagent reinforcement learning in the iterated prisoner's dilemma"*. BioSystems, 37(1,2) :147{66, 1996.

[14]. John Holland. " *les algorithmes génétiques*" . Pour la science n°179.septembre 1992. 44-51.

[17]. J.Greenstette. « genetic algorithms" IEEE. Octobre 1993. 5-8

[20]. DAV92 b LDAVALO. "*Handbook of genetic algorithms*". ED. VNR New York 1992.

[23]. ED Hermes *" algorithmes génétiques et réseaux de neurones application des commandes de processus* ".Bruxelles 1995.

[25]. Z.MICHALEWICZ. "*genetic algorithms + data structures=Evolution programs*". ED.spring-Verlag. New York 1992.

[26]. DREDI Leila. "*Les algorithmes génétiques*". Université de Constantine, 2005.

[27]. Schwefel H.-P., "*Evolution Strategies : A Family of Non-Linear Optimization Techniques Based on Imitating Some Principles of Organic Evolution*", Annals of Operations Research, vol. 1, pp. 165-167, 1984.

[28]. John S. Denker, 1985 "*Dans Les Rêves de la Raison*", Heinz Pagels, pp.118, InterEditions, 1990.

[35]. Khadir Mohamed Tarek, "*Principes de base des réseaux de neurones artificiels et apprentissage*". Support de cours. Université Badji Mokhtar, Annaba.

[37]. Hérault J. et Jutten C., "*Réseaux neuronaux et traitement du signa*"*l*. Ed. Hermès, Paris, 1994.

[38]. Kolen J.F. et Pollack J.B., "*Back-propagation is sensitive to initial conditions*". Technical Report 90-JK-BPSIC, CIS Dept., Ohio St Univ., Columbus, Ohio, 1990.

[39]. Thimm G. et Fiesler E., "*High Order and Multilayer Perceptron Initialization*". IDIAP technical report 94-07, 1994.

[40]. Falhman S.E., "*An Empirical Study of Learning Speed in Backpropagation Network*s". Technical report CMU-CS-88-162, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1988.

[41]. Bottou L-Y., "*Reconnaissance de la parole par réseaux multi-couches*". Proceedings of the International Workshop on Neural Networks and Their Applications, pp 197-217,

1988.

[42]. Lee Y., Oh S.-H. et Kim M.W., "*An Analysis of Premature Saturation in BackPropagation Learning*". Neural Networks, vol. 6, pp. 719-728, 1993.

[43]. Wessels L.F.A. et Barnard E., "*Avoiding False Local Minima by Proper Initialization of Connections*". IEEE Transaction on Neural Networks, vol. 3, N° 6, Novembre 1992.

[44]. Denoeux T. et Lengellé R., "*Initializing Back Propagation Networks With Prototype*"*s*. Neural Networks, Vol. 6, pp 351-363, 1993.

[46]. Belew B., McInerney J. et Schraudolph N., "*Evolving Networks : Using the Genetic Algorithms with Connectionist Learning*". CSE Technical Report CS90-174, Computer Science, UCSD, 1990.

[50]. Axelrod, R. (1987), The evolution of strategies in the iterated prisoner's dilemma, *in* L. D. Davis, ed., "Genetic algorithms and simulated annealing", Morgan Kaufmann.